
Universidade de Aveiro Departamento de Electrónica, Telecomunicações
2008 e Informática

Marco Filipe Nunes Soares Abrantes Pereira

Tecnologia Peer-To-Peer Para Bibliotecas Digitais

Marco Filipe Nunes Soares Abrantes Pereira

Tecnologia Peer-To-Peer Para Bibliotecas Digitais

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Joaquim Arnaldo Carvalho Martins, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Engenheiro Marco Fernandes.

Júri

Presidente

Doutor João Paulo Trigueiros da Silva Cunha
Professor Associado da Universidade de Aveiro

Vogais

Doutor Joaquim Arnaldo Carvalho Martins (Orientador)
Professor Catedrático da Universidade de Aveiro

Doutor Fernando Joaquim Lopes Moreira
Professor Auxiliar do Departamento de Inovação, Ciência e Tecnologia da Universidade Portucalense

agradecimentos Agradeço à minha família que nunca deixou de me apoiar.

Palavras Chave Peer-to-Peer, Biblioteca Digital, JXTA

Resumo As tecnologias peer-to-peer apresentam uma forma interessante de explorar recursos computacionais normalmente desperdiçados. Esta dissertação tem como objectivo avaliar a utilização de tecnologias peer-to-peer como suporte para bibliotecas digitais. Foi utilizada a biblioteca JXTA para criar a rede peer-to-peer. Foi também definida a estratégia para a disponibilização de serviços na rede utilizando um sistema análogo aos tradicionais WebServices baseada na biblioteca JXTA-SOAP.

Keywords Peer-to-Peer, Digital Libraries, JXTA

Abstract Peer-to-peer technologies present an interesting way of exploring usually wasted resources. This dissertation's objective is to evaluate the use of peer-to-peer technologies as a support for digital libraries. The JXTA library was used to create the peer-to-peer network. It was also defined a WebServices like strategy for services availability in the network based on the JXTA-SOAP library.

Índice

Imagens

Lista de Tabelas

Acrónimos

ERP	*Endpoint Routing Protocol*
HTTP	*Hypertext Transfer Protocol*
ID	Identificador
NIO	*Native Input/Output*
PBP	*Pipe Binding Protocol*
PDP	*Peer Discovery Protocol*
PIP	*Peer Information Protocol*
PRP	*Peer Resolver Protocol*
RVP	*Rendezvous Protocol*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
UDP	*User Datagram Protocol*
UUID	*Universally Unique Identifier*
TCP	*Transmission Control Protocol*
WSDL	*Web Service Definition Language*
XML	*Extensible Markup Language*

1 Introdução

Esta dissertação procura explorar a possibilidade da utilização de tecnologias *peer-to-peer* em bibliotecas digitais.

O modelo de rede *peer-to-peer* remonta aos inícios da computação em rede [1] tendo no entanto sido gradualmente preterido em favor do modelo cliente-servidor que está presente actualmente em força na Internet. Nos tempos que correm a imagem natural dos sistemas *peer-to-peer* está ligada ao aparecimento de vários protocolos de transferência de ficheiros como o BitTorrent ou o eDonkey que são utilizados para partilhar diversos conteúdos entre utilizadores de forma rápida e eficaz apesar de muitas das vezes no limite da legalidade [2], o que contribui para uma certa má imagem de todos os sistemas *peer-to-peer*.

O trabalho realizado para esta dissertação envolve vários passos, desde a escolha de uma *framework* de trabalho até a integração de diferentes tecnologias normalmente utilizadas em bibliotecas digitais sobre uma rede *peer-to-peer*. Os exemplos apresentados são sobretudo de nível puramente académico, podendo no entanto e como trabalho futuro, ser plenamente integrados com infra-estruturas já existentes.

1.1 *Motivação*

A utilização de uma rede *peer-to-peer* para suportar as necessidades de armazenamento de informação que uma biblioteca digital apresenta é uma ideia tentadora. Uma rede *peer-to-peer* de partilha de ficheiros representa o conceito de espaço de armazenamento potencialmente ilimitado e completamente distribuído. Se existir a necessidade de mais espaço, simplesmente adiciona-se mais um *peer*. De uma forma geral pode dizer-se que qualquer protocolo *peer-to-peer* já existente poderia satisfazer as necessidades de armazenamento e distribuição de conteúdos que uma biblioteca digital apresenta, no entanto a maioria dos protocolos existentes limita a pesquisa na rede apenas a nomes de ficheiros. Tal limite não é desejável no ambiente em que se inserem as bibliotecas digitais. É neste contexto que surgiu a ideia de integrar uma tecnologia que permita criar

uma rede *peer-to-peer*, o JXTA, com uma tecnologia que permita de forma simples suportar pesquisas ricas, o Apache Lucene. Os índices gerados pelo Lucene adicionam um grau de liberdade sem precedentes às pesquisas que podem ser efectuadas sobre os conteúdos que existem na rede.

A infra-estrutura descrita explora as capacidades de armazenamento dos *peers*, mas desperdiça a sua capacidade de processamento. As bibliotecas digitais necessitam de um conjunto variado de serviços de apoio. Uma rede *peer-to-peer* oferece também uma capacidade de processamento potencialmente ilimitada, que pode e deve ser aproveitada para correr os serviços de suporte necessários. Surge assim a necessidade de combinar a disponibilização de serviços com a rede *peer-to-peer*. Um modelo de disponibilização de serviços que apresenta uma grande popularidade actualmente são os Web Services. Este modelo pode ser adaptado para ser utilizado em conjunto com o JXTA com a utilização da biblioteca JXTA-SOAP, criando assim uma base para a disponibilização de serviços dentro da rede que tem a vantagem de ser construída a custa de protocolos conhecidos.

1.2 *Organização do documento*

Este documento encontra-se organizado em seis capítulos :

1. Introdução – Este capítulo apresenta uma breve introdução ao problema
2. Redes Peer-to-Peer – Este capítulo apresenta uma introdução às redes peer-to-peer e às diferentes configurações que podem adoptar.
3. JXTA – Este capítulo apresenta uma visão detalhada da biblioteca JXTA, que permite a criação de uma rede *peer-to-peer.*
4. Arquitectura P2P para suportar serviços de uma Biblioteca Digital – Este capítulo descreve a possível arquitectura de uma rede *peer-to-peer* que pode ser usada em bibliotecas digitais.
5. Testes – Este capítulo descreve os testes efectuados a uma plataforma *peer-to-peer* desenvolvida com base nas especificações apresentadas no capítulo anterior.
6. Conclusões – Este capítulo apresenta as conclusões gerais do trabalho

2 Redes Peer-to-Peer

Uma rede *peer-to-peer* é um conceito relativamente simples: uma rede em que todos os nós (*peers*) agem como iguais disponibilizando os seus recursos e utilizando os recursos dos outros nós quando necessário[3]. Este é um cenário ideal mas que durante algum tempo principalmente devido à enorme diferença de poder de processamento entre os grandes computadores que foram desenvolvidos para actuarem como servidores e os computadores pessoais de pequeno porte foi substituído por um cenário cliente servidor, cenário esse que predomina actualmente.

Com o evoluir da tecnologia os computadores pessoais foram massificados e atingiram níveis de poder de processamento respeitável de tal forma que o modelo *peer-to-peer* se torna uma alternativa viável ao clássico modelo cliente servidor para a prestação de serviços. O modelo *peer-to-peer* permite pegar em todo o poder de processamento, espaço de armazenamento e recursos variados que existem nas extremidades da rede global e utilizar esse esses recursos de forma construtiva[4]. Basta considerar os ciclos de processamento que são desperdiçados todos os dias para tornar imediatamente atractiva qualquer proposta que rentabilize esses mesmos ciclos.

As redes *peer-to-peer* podem escolher seguir vários caminhos [3],[5] que apresentam diferentes vantagens e desvantagens [6]. As abordagens mais comuns incluem a clássica estrutura totalmente descentralizada em que todos os nós se encontram em posição de igualdade, a abordagem híbrida, em que alguns nós da rede têm mais responsabilidades do que outros, ou até mesmo uma estrutura completamente centralizada onde existe um ou mais servidores centrais que executam tarefas críticas para o bom funcionamento da rede mantendo todo o resto das comunicações numa base *peer-to-per*.

2.1 *Arquitectura centralizada*

Numa arquitectura centralizada existe pelo menos uma máquina com responsabilidades especiais. Tipicamente estas responsabilidades prendem-se com a pesquisa na rede. As redes *peer-to-peer* de partilha de dados são centradas na informação e como tal devem possuir índices que permitam a pesquisa sobre a informação. Se os índices se encontrarem concentrados em *peers* específicos desenhados para lidarem com as pesquisas provenientes de outros *peers* então a rede *peer-to-peer* segue um modelo centralizado. Apesar de a pesquisa ser centralizada o acesso aos dados é feito de forma *peer-to-peer*.

Esta separação entre a informação e os índices que a representam forma uma das bases que permite que esta arquitectura seja facilmente escalável [7] com a introdução de mais *peers* encarregues de lidarem com a indexação e pesquisa. Se os *peers* introduzidos comunicarem entre si cria-se uma topologia híbrida, se os *peers* clientes tiverem de percorrer individualmente os novos *peers* de indexação continuamos a ter uma topologia centralizada. Os *peers* para utilizarem a rede têm de se registar com os *peers* encarregues das pesquisas e transferir para esses *peers* os índices dos seus conteúdos. Esta arquitectura permite localizar dados rapidamente e fornece bases para um controlo de acesso. O tráfego gerado pelas pesquisas é minimizado ao evitar o modelo de *flooding* que aparece muitas vezes nos sistemas totalmente distribuídos ficando no entanto vulnerável a falhas catastróficas dos *peers* que guardam os índices. Para minimizar este possível problema são criados múltiplos *peers* que actuem como servidores de índices, que dependendo da estratégia escolhida podem conter réplicas exactas dos índices a nível global ou apenas dos índices dos *peers* a eles ligados. Dependendo da estratégia utilizada para a transferência dos índices esta arquitectura pode sofrer alguma desactualização nos resultados das pesquisas. Isto acontece quando um *peer* adiciona algo ao seu índice mas não actualiza a versão que transferiu anteriormente.

2.2 *eDonkey*

Este tipo de arquitectura é utilizado no popular protocolo eDonkey implementado em várias aplicações de partilha de ficheiros. Neste protocolo a primeira acção de um *peer* deve ser ligar-se a um servidor cuja principal missão é indexar os ficheiros disponíveis na rede. A ligação ao servidor é mantida aberta durante todo o tempo de permanência do *peer* na rede. Apesar de actualmente existirem muitos servidores disponíveis ao público estes não comunicam entre si [8] (o cliente pode utilizar ligações UDP adicionais para efectuar pesquisas em servidores que não aquele ao qual se encontra ligado). As pesquisas são efectuadas sobre os índices existentes no servidor e devolvem listas de *peers* (*sources*) que possuam ficheiros que correspondam ao critério da pesquisa. Para efectuar o *download* de um ficheiro são estabelecidas ligações directas com as várias *sources*, podendo o *download* começar imediatamente ou ficar numa fila de espera.

2.2.1 ICQ

A versão original do ICQ utiliza uma arquitectura *peer-to-peer* centralizada. Existe um servidor central responsável por vigiar o estado dos clientes (*peers*) e notificar todos os interessados da mudança de estado de cada cliente. Quando um cliente quer contactar com outro que se encontre disponível não é necessária a intervenção do servidor na comunicação criando assim a parte *peer-to-peer* da arquitectura [9].

2.3 *Arquitectura descentralizada*

A Figura 1 ilustra um cenário típico de uma rede totalmente descentralizada. Neste tipo de redes todos os nós se encontram em posição de igualdade mas possivelmente nem todos os nós se conhecem. Se todos disponibilizarem os mesmos serviços seria possível remover arbitrariamente qualquer nó da rede e não sofrer qualquer quebra no acesso aos serviços oferecidos. Este tipo de rede permite um crescimento potencialmente ilimitado do número de nós presentes na rede no entanto quanto maior a rede maior será a probabilidade de nós nos

extremos da rede não se conhecerem mutuamente o que pode levar a um menor número de resultados do que os que realmente existem numa pesquisa. Dado que não existe nenhum *peer* em particular responsável por manter um índice dos conteúdos existentes na rede esta tarefa é da responsabilidade de todos os *peers*. Para efectuar uma pesquisa na rede normalmente é utilizado um processo de *flooding* em que cada *peer* contacta todos os *peers* que conhece que por sua vez vão contactar outros *peers* que conheçam. É necessário introduzir um limite para a propagação das mensagens de pesquisa para evitar que circulem eternamente na rede. Esse limite é definido normalmente à custa do número de saltos que uma mensagem pode efectuar antes de um *peer* deixar de a reencaminhar para todos os que conhece. Esta estratégia de *flooding* gera uma quantidade enorme de tráfego [7] e é uma das forças motrizes da evolução deste tipo de sistemas. Um problema que existe neste tipo de arquitectura é o do acesso à rede. Como não existe um ponto central torna-se difícil controlar o acesso à rede. Esta característica torna este tipo de redes muito resistente à censura e a ataques já que para neutralizar uma rede com tamanho considerável é necessário um ataque em larga escala a múltiplos alvos ao mesmo tempo, sem qualquer garantia de que faça mais do que isolar partes da rede. Para aceder a rede é necessário conhecer um *peer* que já faça parte da rede, o que pode ser um problema. Para resolver este problema são normalmente criados *peers* públicos aos quais qualquer outro *peer* se possa ligar.

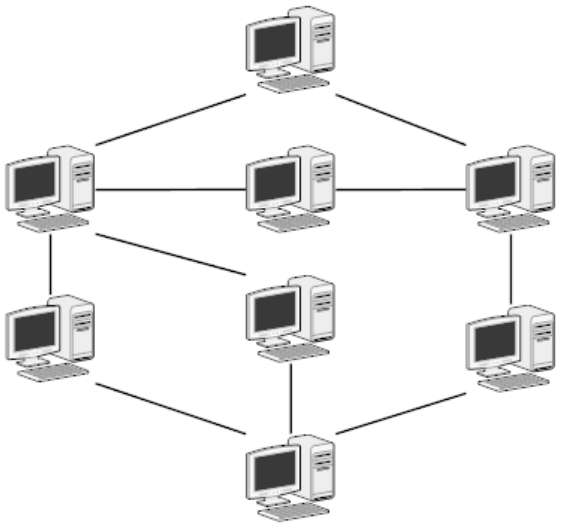

Figura 1: Rede *peer-to-peer* descentralizada

2.3.1 Gnutella v0.4

Este tipo de arquitectura é utilizado na versão 0.4 do protocolo Gnutella[10]. Existem listas de *peers* públicos aos quais um *peer* se pode ligar para começar a utilizar a rede[3]. Quando é efectuada uma pesquisa na rede esta é enviada para todos os *peers* aos quais o *peer* que efectua a pesquisa esteja directamente ligado. Os *peers* que recebem uma pesquisa propagam-na a não ser que esta tenha atingido o seu limite de saltos máximo (designado por horizonte e cujo valor é normalmente de sete saltos) caso no qual é descartada. Os resultados da pesquisa são devolvidos ao *peer* que iniciou a pesquisa pelo caminho inverso ao que seguiram ao ser propagados. Uma transferência de um ficheiro segue pelo caminho que a pesquisa que lhe deu origem seguiu até encontrar o *peer* que possui o ficheiro pretendido. Se este *peer* aceitar a transferência então liga-se directamente ao *peer* que a requisitou [10].

2.4 *Arquitectura híbrida*

A Figura 2 ilustra uma rede *peer-to-peer* híbrida. Com este tipo de redes procura-se combinar as vantagens das duas arquitecturas anteriores. As arquitecturas híbridas resultam normalmente da evolução natural que ocorre nos dois tipos de arquitecturas vistos anteriormente. Quando são acrescentados servidores extra que comunicam entre si a uma arquitectura centralizada passamos a ter uma arquitectura híbrida. Por outro lado quando se reconhece que numa rede com topologia totalmente distribuída existem nós que devido ao seu posicionamento privilegiado vão estar ligados a mais *peers* que a maioria e se atribui maior responsabilidade a esses nós cria-se também uma arquitectura híbrida. É interessante constatar que arquitecturas que partem de bases diferentes convergem para a mesma solução [3] (mantendo no entanto características próprias).

Os nós que aceitam maiores responsabilidades neste tipo de redes designam-se por "*Super Peers*". O modo de funcionamento deste tipo de redes varia de acordo com a arquitectura que a rede possuía anteriormente, mas em traços gerais a introdução de *Super Peers* afecta de forma positiva a as pesquisas efectuadas na

rede actuando como ponto de encontro para os restantes *peers* (o que contribui para minimizar o fenómeno de isolamento de *peers* que pode acontecer nas arquitecturas totalmente distribuídas). A introdução de múltiplos *Super Peers* contribui para evitar a existência de um ponto de falha único, um problema que afecta as arquitecturas centralizadas.

Os *Super Peers* podem ser escolhidos de forma estática e colocados a disposição do resto da rede da mesma forma que os servidores numa arquitectura centralizada ou de forma dinâmica onde um *peer* dependendo das suas características e qualidade da ligação a rede se pode voluntariar para receber mais ligações que os restantes tornando-se um *Super Peer*. Uma falha num *Super Peer* pode conduzir ao isolamento de parte da rede (ao contrário da topologia centralizada onde uma falha no servidor conduziria eventualmente a uma falha total da rede).

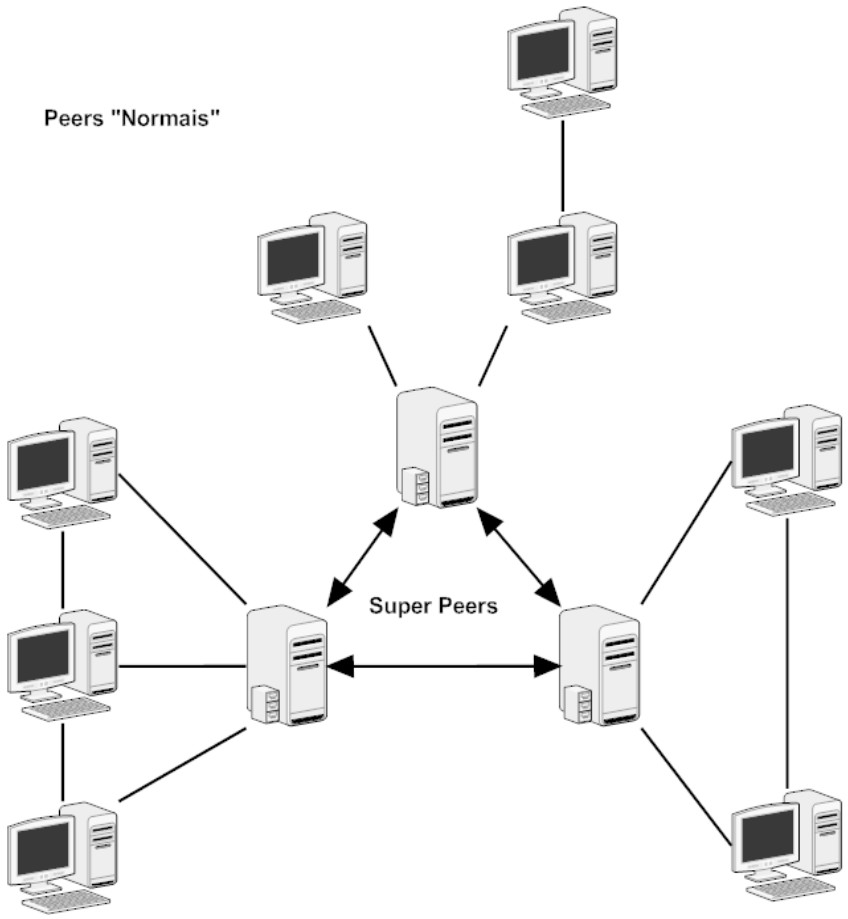

Figura 2: Rede *peer-to-peer* híbrida

2.4.1 Gnutella v0.6

A versão 0.6 [11] do protocolo Gnutella introduz o conceito de "*Ultrapeers*" e "*Leaf Peers*"[12] criando uma arquitectura híbrida a partir de uma arquitectura distribuída. É criada uma separação hierárquica entre os *peers* que se espera que não estejam ligados muito tempo a rede ou cujas suas capacidades possam atrasar o processo de propagação de pesquisas (que permanece essencialmente igual ao praticado na versão 0.4 do protocolo) e os *peers* que pelas suas capacidades sejam úteis ao processo de pesquisa. Os *Ultrapeers* são utilizados pelos *Leaf Peers* para acederem a rede, num processo que mimetiza a organização da própria Internet [13]. Um *Leaf Peer* é encorajado a ter um pequeno número de ligações a outros *peers*, preferencialmente a *Ultrapeers* enquanto um *Ultrapeer* é encorajado a manter ligações a muitos *Leaf Peers* e a alguns *UltraPeers* passando a agir como um *proxy* de acesso a rede para os *Leaf Peers* aos quais está ligado.

Dado que o Gnutella seguia originalmente uma arquitectura completamente descentralizada colocou-se a questão de como eleger os *peers* que seriam promovidos a *Ultrapeers*. A solução escolhida foi permitir que todos os *peers* que respeitassem um conjunto de critérios [11] possam ser promovidos a *Ultrapeers* consoante as necessidades da rede.

2.5 *Comparação entre arquitecturas*

A Tabela 1 apresenta uma comparação entre as arquitecturas referidas anteriormente. São utilizados três critérios para comparar as arquitecturas: escalabilidade que avalia a capacidade de a rede crescer, a robustez que avalia a capacidade de a rede resistir a falhas de peers críticos e o acesso que avalia o quão difícil para um nó ligar-se à rede e obter resultados significativos de um a pesquisa. Cada um destes critérios é classificado de 1 a 5, sendo 1 a classificação mais baixa e 5 a mais alta.

Arquitectura	Escalabilidade	Robustez	Acesso
Centralizada	2	1	5
Descentralizada	5	5	1
Híbrida	4	4	4

Tabela 1: Comparação entre arquitecturas

As arquitecturas centralizadas ao possuírem um ponto central de falha são claramente as menos robustas, enquanto as arquitecturas descentralizadas são as mais robustas devido a falta desse ponto fraco. As arquitecturas híbridas dependem do número de *super peers* para criar uma rede robusta. Com um só *super peer* esta arquitectura encontra-se ao nível da arquitectura centralizada, mas à medida que vão sendo adicionados *super peers* a robustez da rede aumenta. Por outro lado os *super peers* contribuem para que as arquitecturas híbridas sejam bastante escaláveis e tornam o acesso a rede tão fácil como se esta fosse uma rede com uma arquitectura centralizada. O acesso à rede é o ponto forte das arquitecturas centralizadas e o ponto mais fraco das arquitecturas descentralizadas. As arquitecturas descentralizadas possuem um potencial de crescimento praticamente ilimitado, potencial esse que é conseguido à custa de uma também potencial perda de resultados nas pesquisas e da dificuldade que um peer pode ter para se juntar à rede.

A arquitectura híbrida ao aproveitar os melhores elementos da arquitectura centralizada e descentralizada torna-se a arquitectura de eleição para a criação de redes *peer-to-peer*.

3 JXTA

O projecto JXTA [14] é um projecto *open source* que visa criar uma plataforma *peer-to-peer* de uso geral que pode ser utilizada tanto para a partilha de recursos como para a disponibilização de serviços [15]. JXTA não é um acrónimo críptico é uma abreviatura da palavra inglesa *Juxtapose*, literalmente justaposto e vem do objectivo do projecto não ser criar uma alternativa à arquitectura cliente servidor, mas sim de encontrar uma forma de as arquitecturas *peer-to-peer* e cliente servidor conseguirem trabalhar lado a lado (justapostas) de forma a atingirem um objectivo comum.

O JXTA é um conjunto de protocolos que forma o cerne de uma plataforma *peer-to-peer* sobre a qual podem ser construídos diversos tipos de serviços. Os protocolos foram desenhados de forma a poderem ser implementados em qualquer linguagem de programação, o que permite o aparecimento de uma rede composta de diversos tipos de dispositivos. De forma a alcançar uma independência da plataforma utilizada as mensagens no JXTA são definidas e transmitidas em XML [16] [17]. Para além das mensagens o XML é utilizado também nos anúncios (*advertisements*) que servem de suporte a plataforma. Estes *advertisements* podem ser utilizados para descrever diversos tipos de recursos ou serviços que possam estar a disposição dos *peers*.

Os mecanismos primários de comunicação no JXTA são os pipes, canais unidireccionais por onde as mensagens são enviadas.

A arquitectura definida pela especificação do JXTA é composta por três camadas:

- *Core Layer* – Esta camada é responsável pela implementação dos protocolos base que fazem toda a plataforma funcionar em conjunto. Estes protocolos são responsáveis pela criação e gestão de *Peer Groups*, mecanismos de comunicação e transporte de mensagens e mecanismos básicos de segurança.
- *Service Layer* – Esta camada utiliza os protocolos da camada anterior de forma a produzir serviços utilizáveis pelas aplicações.

- *Application Layer* – Esta camada utiliza os serviços implementados na camada anterior e disponibilizados por vários *peers* para construir uma aplicação completa.

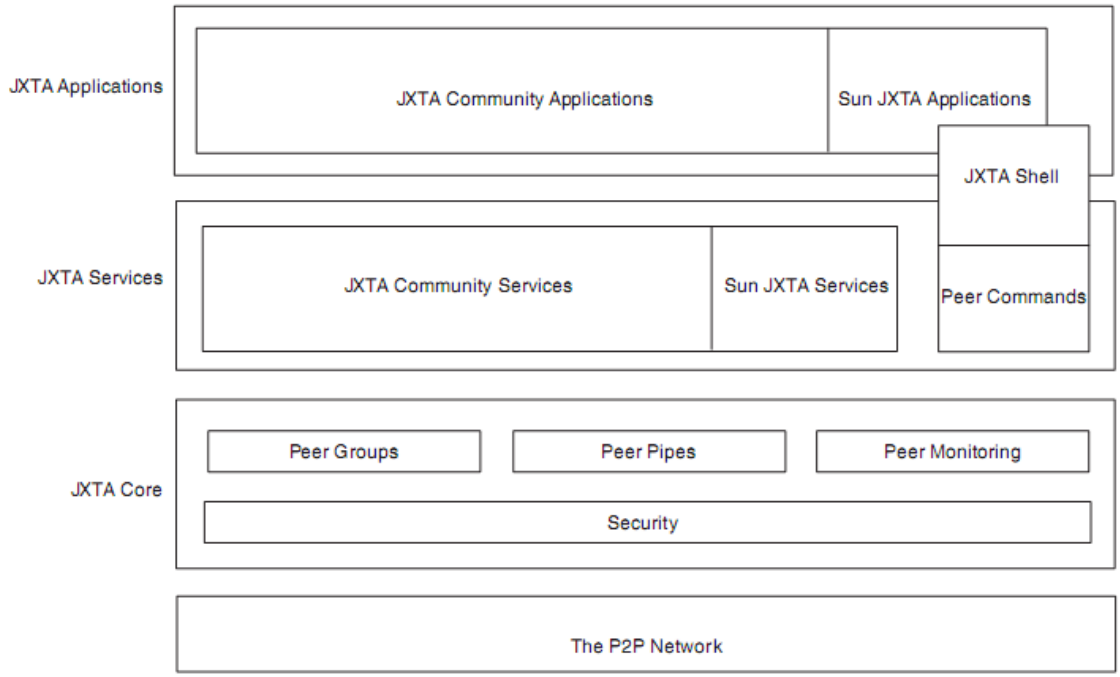

Figura 3: Arquitectura geral do JXTA [9]

3.1 *Peers no JXTA*

Um aspecto fundamental das redes *peer-to-peer* é a própria definição do que é um *peer*. Para o JXTA um *peer* é um qualquer dispositivo ou aplicação com acesso a uma rede que implemente um ou mais dos seus protocolos nucleares [15]. Isto faz com que tanto um telemóvel como um computador de secretária possam comunicar de forma quase transparente, e com que um único dispositivo físico possa albergar vários *peers*. Todos os *peers* no JXTA possuem um identificador único que é usado principalmente para encaminhamento de mensagens. A comunicação entre *peers* pode não ser feita de forma directa, devido a existência de diferentes barreiras entre redes. Para permitir a transposição de tais barreiras os *peers* podem utilizar os serviços de *peers* especializados, como *rendezvous*, *relay* ou *proxy peers*. Isto cria automaticamente várias categorias de *peers*.

Os *peers* no JXTA podem ser configurados (se tiverem implementado os protocolos para tal) para descobrir automaticamente outros *peers* e para formarem grupos de *peers* com interesses comuns.

3.2 *Tipos de peers*

Os *peers* no JXTA podem ser genericamente classificados em três categorias básicas: *Minimal Edge*, *Full Edge*, e *Super Peers*. As responsabilidades de cada categoria no funcionamento da rede são diferentes, com um *Minimal Edge* a ter o menor grau de responsabilidade e os *Super Peers* a ter o maior. Dentro da categoria dos *Super Peers* existem três papeis que um *peer* pode assumir: *Relay*, *Rendezvous* e *Proxy*. Nada impede que um único *Super Peer* assuma uma combinação dos papéis disponíveis, ou seja um Super Peer pode ter simultaneamente funções de *Relay* e *Rendezvous* se for configurado para tal.

- *Minimal Edge Peer* – Este tipo de *peers* implementa apenas as funcionalidades mínimas necessárias para poder ser considerado um *peer* pela definição do JXTA. Requerem normalmente a ajuda de um *Super Peer* do tipo *Proxy* para poderem aceder e utilizar correctamente os recursos disponíveis na rede. Podem enviar e receber mensagens, mas não possuem normalmente recursos para fazer cache dos diversos tipos de *advertisements* utilizados pela rede, o que faz com que não respondam a pedidos de descoberta de outros *peers*.
- *Full Edge Peer* – Este tipo de *peers* implementa todas as funcionalidades necessárias para aceder e utilizar a rede sem a ajuda de um *Proxy Peer.* Este tipo de *peers* forma a espinha dorsal da rede. Possui recursos suficientes para fazer cache dos *advertisements* utilizados pela rede, o que quer dizer que pode responder a pedidos de descoberta de outros *peers*, mas não retransmite esses pedidos para outros *peers* que conheça.
- *Super Peer* – Este tipo de *peers* formam a infra-estrutura de apoio a rede. Não são estritamente necessários para o funcionamento da rede (por exemplo em redes locais que possuam mecanismos de *broadcast/multicast* os *Edge Peers* podem descobrir-se uns aos outros sem

a ajuda de um *Rendezvous*). Os *Super Peers* podem ser divididos em três categorias com base no papel que desempenham na rede:

- *Rendezvous* – Este tipo de *Super Peer* é responsável pela propagação de mensagens pela rede. Os *Rendezvous* são responsáveis por manter um índice global de *advertisements* de forma a poderem responder de forma eficiente a diversos tipos de pedidos. Serve também como *peer* de suporte a propagação de mensagens, visto que quando não existe um caminho directo entre dois outros *peers* as mensagens podem ser passadas de *Rendezvous* em *Rendezvous* até atingirem o destino.
- *Relay* – Este tipo de *Super Peer* é utilizado para ajudar a transpor barreiras entre redes. Possui mecanismos de *caching* de mensagens que podem ser utilizados por outros *peers* para contactar *peers* que de outra forma seriam incontactáveis
- *Proxy* – Este tipo de *Super Peer* é utilizado como ponto de acesso a rede pelos *Minimal Edge Peers*. Ele possui os recursos de *cache* que os *Minimal Edge Peers* não possuem, podendo assim responder aos pedidos feitos por outros *peers*. Se tal for necessário pode ser utilizado para traduzir mensagens enviadas pelos *Minimal Edge Peers* para um formato que os outros membros da rede compreendam, ou para fazer pedidos específicos de protocolos que os *Minimal Edge Peers* não implementem em nome deles.

3.3 *Peer Groups*

Um *peer* sozinho é apenas um recurso por explorar. Por outro lado se esse *peer* se juntar a um grupo constituído por outros *peers* que tenham interesse nos serviços que ele tenha para oferecer ou cujos serviços ele queira aproveitar então esse grupo passa ser um recurso valioso. O JXTA oferece mecanismos para que os *peers* se organizem em grupos de forma simples e conveniente. Estes mecanismos existem para que seja uma tarefa acessível a um *peer* relativamente simples criar, ou descobrir e juntar-se a um grupo (que é essencialmente instanciar todos os serviços requeridos pelo grupo a que se queira juntar), sem

qualquer interferência nos motivos que levam um *peer* a juntar-se ou a criar um grupo.

Todos os *peers* começam por fazer parte de um *peer group* universal designado por *World Peer Group*. Este grupo fornece apenas serviços mínimos de propagação de mensagens dentro do grupo e as aplicações normalmente não interagem com ele. Existe um segundo *peer group* designado por *Network Peer Group* ao qual os peers se juntam que esse sim fornece serviços de mais alto nível. É deste *Network Peer Group* que derivam os grupos privados que as aplicações podem criar. Cada grupo é responsável por implementar dois serviços base, e cinco outros serviços que apesar de opcionais são definidos como *standard* pela implementação actual do JXTA [16]:

- *Endpoint Service* – Este serviço é utilizado para encaminhar mensagens entre peers.
- *Resolver Service* – Este serviço é utilizado para enviar *querys* genéricas entre *peers*.
- *Discovery Service* – Este serviço é responsável pela descoberta de conteúdo dentro de um grupo.
- *Membership Service* – Este serviço forma a base para a implementação de grupos que necessitam de segurança ao nível dos seus membros. É utilizado para estabelecer identidades de *peers* dentro do grupo.
- *Access Service* – Este serviço é responsável por validar os pedidos dos peers que necessitem de autorização para serem processados.
- *Pipe Service* – Este serviço é responsável pela criação de pipes. Os pipes criados num *peer group* só tem significado nesse grupo.
- *Monitoring Service* – Este serviço permite que os *peers* monitorizem o estado do grupo e de outros membros do grupo.

Como normalmente os grupos privados derivam do *Network Peer Group* estes serviços já se encontram disponíveis.

3.4 *Modules*

Os protocolos do JXTA permitem a criação de uma rede funcional, mas é normalmente necessário acrescentar funcionalidades extra de forma a tornar a rede verdadeiramente útil. Para tal o JXTA introduz o conceito de *module*, que é essencialmente uma abstracção sobre um pedaço de código e a interface necessária para o utilizar. Esta abstracção permite criar uma rede de serviços verdadeiramente heterogénea em que a mesma funcionalidade é implementada de formas completamente distintas por *peers* distintos. Para que esta abstracção global funcione são definidas três outras abstracções:

Module Class – Esta abstracção serve essencialmente para formalizar a existência de um determinado comportamento ou funcionalidade na rede.

Module Specification – Esta abstracção serve para aceder a funcionalidade que é fornecida por um módulo, e deve conter toda a informação necessária para a interacção com o módulo, ou serviço.

Module Implementation – Esta abstracção serve para detalhar a existência de uma implementação específica de uma funcionalidade descrita por um *Module Specification.*

3.5 *Advertisements*

Todos os recursos presentes numa rede JXTA são representados por *advertisements*, que são pequenos documentos XML. Os *peers* publicam na rede um *advertisement* a descrever um recurso que pretendam dar a conhecer a rede (por exemplo um pipe de comunicação), e os outros restantes *peers* pesquisam sobre os *advertisements* de forma a encontrarem esse pipe, criando assim um mecanismo uniforme de divulgação de recursos. Os protocolos do JXTA definem oito tipos de *advertisements*:

- *Peer Advertisement* – Este *advertisement* é utilizado para descrever as características de um *peer*. Incluem normalmente o ID na rede e o nome do peer.

- *Peer Group Advertisement* – Este *advertisement* é utilizado para descrever um *peer group*. Inclui o nome e o ID do grupo, tal como os parâmetros necessários para que um *peer* possa entrar para o grupo.
- *Pipe Advertisement* - Este *advertisement* é utilizado para descrever um pipe. Este tipo de *advertisement* é essencial para o estabelecimento de canais de comunicação entre *peers*. Inclui o nome, o ID, e o tipo do pipe.
- *Module Class Advertisement* – Este *advertisement* é utilizado para divulgar pela rede a existência de uma funcionalidade. Inclui um ID, e um nome.
- *ModuleSpec Advertisement* – Este *advertisement* é utilizado para definir uma especificação de um serviço ou funcionalidade. É também necessário quando se quer utilizar um serviço remoto. Inclui o nome, o ID e um conjunto de parâmetros que devem ser interpretados pela implementação do serviço.
- *ModuleImpl Advertisement* – Este *advertisement* é utilizado para divulgar os detalhes específicos de uma implementação de um module. Inclui o nome, tal como o ID do *ModuleSpec* que implementa e outros parâmetros necessários.
- *Rendezvous Advertisement* – Este *advertisement* é utilizado para descrever um *peer* que actua como *rendezvous* dentro de um grupo.
- *Peer Info Advertisement* – Este *advertisement* é utilizado para anunciar o estado de um *peer*.

Para além destes tipos pré-definidos é possível definir novos tipos de *advertisements* quer utilizando os tipos já existentes como base, quer criando tipos inteiramente novos.

3.6 *Mensagens*

Uma mensagem no JXTA é a unidade básica de transferência de informação entre *peers*. Os protocolos base do JXTA são definidos a custa de mensagens. Para todos os efeitos as mensagens são documentos XML cujos elementos podem ser definidos pelas aplicações criando desta forma protocolos adaptáveis. Estas são as mensagens com as quais os utilizadores lidam.
Internamente para a transmissão pela rede o JXTA define representações destas mensagens sob dois formatos: um formato binário e um formato XML. Estas representações foram desenhadas para se adequarem ao protocolo de transporte real (TCP, HTTP) que está a ser utilizado.

3.7 *Protocolos Base*

Existem seis protocolos base no JXTA. Estes protocolos são conjuntos de mensagens XML, utilizados para as funcionalidades mais básicas da plataforma, tais como descoberta de *peers* e encaminhamento de mensagens. Os protocolos definidos seguem um modelo do tipo *query/response*, no qual não há garantias do número de respostas que um *peer* vai obter quando faz uma qualquer *query*. Um *peer* não tem de implementar todos os protocolos, apenas os que planeia utilizar. Estes protocolos não foram pensados para ser usados directamente pelos utilizadores do JXTA (apesar de ser possível utilizar estes protocolos para implementar funcionalidades), mas constituem a base de tudo o que pode ser construído com o JXTA. A relação entre os protocolos pode ser vista na Figura 4.

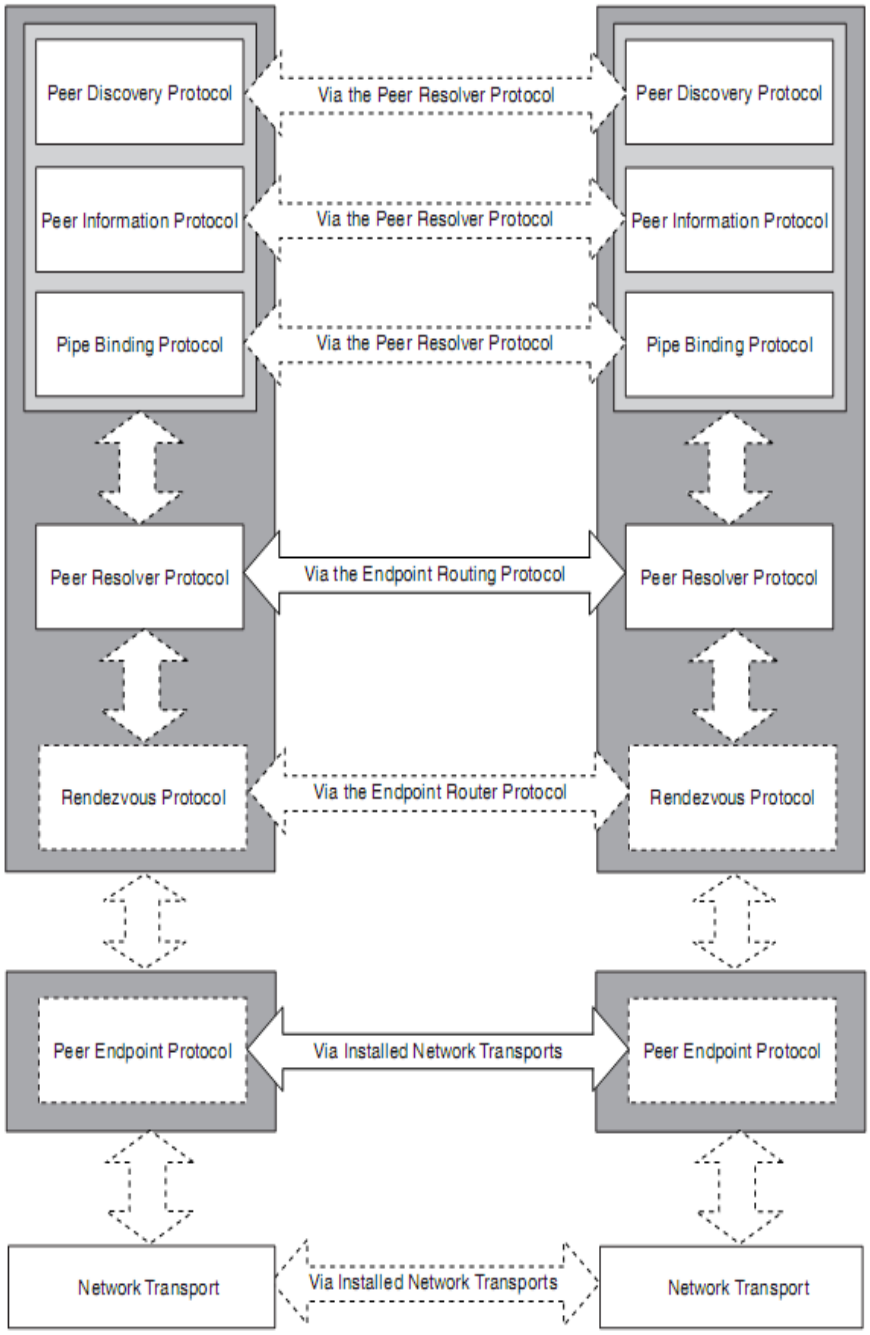

Figura 4: Pilha Protocolar do JXTA [9]

3.7.1 *Endpoint Routing Protocol* (ERP)

Este protocolo é utilizado para criar a ilusão de uma rede complemente conexa, mesmo quando esta não existe. Utilizando o ERP os *peers* conseguem descobrir rotas de forma a comunicarem uns com os outros. Quando um *peer* pretende

comunicar com outro e não possui uma rota para o alcançar deve enviar uma mensagem de *Route Query*, a qual todos os *peers* podem responder, se tiverem conhecimento para tal. A resposta vem sob a forma de uma mensagem de Route *Response*. As mensagens de *Route Query* (Figura 6) e *Route Response* (Figura 7) são encapsuladas em *queries* do PRP. Para que este protocolo funcione as rotas tem de ser publicitadas utilizando o mecanismo de *advertisements* do JXTA, formando um *Route Advertisement* com um *schema* que pode ser visto na Figura 5. O ERP pode também ser utilizado como protocolo de transporte de mensagens.

```
<xs:element name="APA" type="jxta:APA"/>
<xs:complexType name ="jxta:APA">
  <xs:sequence>
    <xs:element name="PID" minOccurs="0" type="jxta:JXTAID"/>
    <xs:element name="EA" type="xs:anyURI" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="RA" type="jxta:RA"/>
<xs:complexType name ="jxta:RA">
  <xs:sequence>
    <xs:element name="DstPID" minOccurs="0" type="jxta:JXTAID"/>
    <xs:element name="Dst">
      <xs:sequence>
        <xs:element ref="jxta:APA" maxOccurs="1"/>
      </xs:sequence>
    </xs:element>
    <xs:element name="Hops" minOccurs="0">
      <xs:sequence>
        <xs:element ref="jxta:APA" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figura 5: XML *Schema* de um *Route Advertisement*

Significado de cada elemento:

- `<PID>` – A identificação do *peer* descrito no *advertisement*.
- `<EA>` – Um identificador de um *endpoint* pertencente ao *peer*.
- `<DstPID>` – A identificação do *peer* descrito no *advertisement*.
- `<Dst>` – Este elemento representa uma lista de identificadores de *endpoints* pertencentes ao *peer*.

- <Hops> – Este elemento representa uma lista de identificadores de *endpoints* que podem ser utilizados como rota para aceder ao *peer* descrito pelo elemento <Dst>. Os *peers* encaminham as mensagens para o primeiro *peer* pertencente a esta lista que reconheçam.

É importante notar que existem elementos com informação potencialmente duplicada, como os elementos <PID> ou <DstPID>. Para evitar essa redundância estes elementos podem ser omitidos quando a informação que representam já é conhecida (no caso do elemento <PID> acontece quando é utilizado no elemento <Dst> de um *Route Advertisement*).

```
<xs:element name="ERQ" type="jxta:ERQ"/>
<xs:complexType name ="jxta:ERQ">
  <xs:sequence>
    <xs:element name="Dst" type="jxta:JXTAID"/>
    <xs:element name="Src">
      <xs:element ref="jxta:RA"/>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figura 6: XML *Schema* de uma *Route Query*

Significado de cada elemento:

- <Dst> – A identificação do *peer* para o qual se quer obter uma rota.
- <Src> – Um *Route Advertisement* que contem uma rota para o *peer* que requisitou a informação.

```
<xs:element name="ERR" type="jxta:ERR"/>
<xs:complexType name ="jxta:ERR">
  <xs:sequence>
    <xs:element name="Dst">
      <xs:element ref="jxta:RA"/>
    </xs:element>
    <xs:element name="Src">
      <xs:element ref="jxta:RA"/>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figura 7: XML Schema de uma Route Response

Significado de cada elemento:

- `<Dst>` – Este elemento representa o *Route Advertisement* do *peer* para o qual foi solicitada uma rota numa mensagem anterior de *Route Query*.
- `<Src>` – Um *advertisement* que contem uma rota para o *peer* que requisitou a informação.

Quando usado como protocolo de transporte e para garantir que uma mensagem enviada chega ao destino o ERP acrescenta um elemento as mensagens que transporta, elemento esse que pode ser visto na Figura 8

```
<xs:element name="ERM" type="jxta:ERM"/>

<xs:complexType name="ERM">
  <xs:sequence>
    <xs:element name="Src" type="jxta:JXTAID" />
    <xs:element name="Dest" type="xs:anyURI" />
    <xs:element name="LastHop" minOccurs="0" type="jxta:JXTAID" />
    <xs:element name="Fwd">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="jxta:APA" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="Rvs" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="jxta:APA" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figura 8: XML *Schema* de um *EndpointRouter Message Element*

Significado de cada elemento:

- `<Src>` – A identificação do *peer* gerou a mensagem.
- `<Dest>` – O *endpoint* do *peer* final que deve receber a mensagem.
- `<LastHop>` – A identificação do último *peer* que reenviou a mensagem. Cada *peer* que reenvie a mensagem deve actualizar o valor deste elemento para corresponder a sua própria identificação.
- `<Fwd>` – Este elemento descreve uma lista de *peers* que podem ser utilizados para fazer a passagem de mensagens entre a origem e o

destino. É uma lista que pode conter diferentes rotas. De forma a enviar a mensagem, um *peer* deve percorrer esta lista a procura do último *peer* que reconheça e reenviar a mensagem para ele.

- `<Rvs>` – Este elemento descreve uma lista *peers* que formem a rota inversa a que a mensagem está a percorrer (se estiver a seguir uma rota do *peer* A para o *peer* B este elemento descreve uma possível rota de B para A).

Este protocolo define ainda uma mensagem adicional, o NACK, que é enviada para um *peer* quando é detectado que ele está a utilizar uma rota que já não é valida para tentar comunicar com outro *peer*.

3.7.2 Rendezvous Protocol (RVP)

Este protocolo é utilizado para controlar a propagação de mensagens num *peer group*, para que as mensagens enviadas para múltiplos *peers* não formem ciclos de encaminhamento nem sejam propagadas eternamente pela rede. Os protocolos de transporte que o JXTA normalmente suporta não foram desenhados para enviar informação de uma única fonte para múltiplos receptores ao mesmo tempo, (com a excepção do suporte para IP *multicast* nas redes locais), o que criou a necessidade de criar um protocolo para o conseguir. Um exemplo do uso deste protocolo é a pesquisa por a*dvertisements*, que podem utilizar o ERP ou o RVP.

Este protocolo está associado ao *super peer* do tipo *Rendezvous*. Este tipo de *peers* difunde pela rede *advertisements* específicos, os *Rendezvous Advertisements* como o da Figura 9 de forma a poderem ser encontrados por outros *peers*. Um conjunto de *rendezvous peers* que coopere consegue construir uma *Peer View*, que é nada mais do que uma lista de *peers* que naquele momento se encontram a actuar como um *rendezvous*, que pode ser utilizada para difundir mensagens pela rede de forma eficiente.

```
<xs:element name="RdvAdvertisement" type="jxta:RdvAdvertisement"/>
<xs:complexType name="RdvAdvertisement">
  <xs:sequence>
    <xs:element name="RdvGroupId" type="jxta:JXTAID" />
    <xs:element name="RdvPeerId" type="jxta:JXTAID" />
    <xs:element name="RdvServiceName" type="xs:string" />
    <xs:element name="Name" type="xs:string" />
    <xs:element name="RdvRoute" type="jxta:RA" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Figura 9: XML *Schema* de um *Rendezvous Advertisement*

Significado de cada elemento:

- `<RdvGroupID>` – Este elemento contem um identificador do *peer group* do qual este *peer* está a actuar como *Rendezvous*.
- `<RdvPeerId>` – Este elemento contem o identificador do *peer* que criou este *advertisement* e que está a actuar como *Rendezvous*.
- `<RdvServiceName>` – Este elemento identifica o *PeerView* ao qual este *Rendezvous* pertence.
- `<Name>` – O nome pelo qual é conhecido este *Rendezvous*. É normalmente igual ao nome do *peer*.
- `<RdvRoute>` – Este elemento opcional contem um *Route Advertisement* que pode ser utilizado para contactar este *Rendezvous*.

De forma a controlar a propagação de mensagens enviadas por este mecanismo, é acrescentado a cada mensagem enviada um elemento de controlo cujo *schema* é o da Figura 10

```
<xs:element name="RendezVousPropagateMessage"
type="jxta:RendezVousPropagateMessage"/>
<xs:complexType name="RendezVousPropagateMessage">
  <xs:sequence>
    <xs:element name="MessageId" type="xs:string" />
    <!-- This should be a constrained subtype -->
    <xs:element name="DestSName" type="xs:string" />
    <xs:element name="DestSParam" type="xs:string" />
    <xs:element name="TTL" type="xs:unsignedInt" />
    <xs:element name="Path" type="jxta:JXTAID" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Figura 10: XML *Schema* de um *Rendezvous Advertisement*

Significado de cada elemento:

- `<MessageId>` – O identificador associado à mensagem. Quando um *peer* recebe uma mensagem com um identificador que já viu não a propaga mais.
- `<DestSName>` – Este elemento deve conter o nome do destino da mensagem
- `<DestSParam>` – Este elemento deve conter os parâmetros necessários para que o destino da mensagem a aceite.
- `<TTL>` – Este elemento controla o número de *peers* que a mensagem ainda pode percorrer. Quando chegar a zero a mensagem não deve ser propagada.
- `<Path>` – Este elemento representa os *peers* pelos quais esta mensagem já passou. Se a mensagem regressar de novo a um *peer* presente num destes elementos então é porque existe um ciclo, e a mensagem não deve ser propagada de novo.

3.7.3 Peer Resolver Protocol (PRP)

Este protocolo é utilizado para enviar e receber querys genéricas. As querys são válidas dentro de um determinado peer group. É um protocolo desenhado para ser utilizado como base comum pelos outros protocolos como o PDP ou o PIP, de forma a evitar uma profusão de mensagens específicas para cada protocolo. Dado que forma a espinha dorsal dos restantes protocolos é necessária uma referência ao formato das suas mensagens:

```
xs:element name="ResolverQuery" type="jxta:ResolverQuery"/>
<xs:complexType name="ResolverQuery">
  <xs:sequence>
    <xs:element ref="jxta:Cred" minOccurs="0" />
    <xs:element name="SrcPeerID" type="jxta:JXTAID" />
    <xs:element name="HandlerName" type="xs:string" />
    <xs:element name="QueryID" type="xs:string" />
    <xs:element name="HC" type="xs:unsignedInt" />
    <xs:element name="Query" type="xs:anyType" />
  </xs:sequence>
</xs:complexType>
```

Figura 11: XML Schema de uma Query do PRP

Significado de cada elemento:

- `<jxta:Cred>` – Uma credencial de identificação do *peer*.
- `<SrcPeerID>` – A identificação do *peer* que enviou a *query*
- `<HandlerName>` – Uma string utilizada para decidir como e por quem deve ser interpretado o que aparecer contido no elemento *query*.
- `<QueryID>` – Um identificador que deve ser colocado nas respostas a *querys* para permite ligar uma *query* à sua resposta.
- `<HC>` – O número de *peers* pelo qual esta *query* já passou. Cada *peer* que reenvia esta *query* deve incrementar este campo de forma ser possível controlar o quantos saltos uma *query* deve percorrer antes de deixar de ser reenviada.
- `<Query>` – A *query* propriamente dita, representada sob a forma de uma string que o destinatário entenda. Desta forma o conteúdo pode ser outra mensagem XML de um protocolo de nível superior.

Na Figura 12 pode ver-se o aspecto que uma *query* do PRP tem. O conteúdo do elemento `<Query>` foi omitido para abreviar a representação.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:ResolverQuery>
<jxta:ResolverQuery xmlns:jxta="http://jxta.org">
  <HandlerName>
    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000305
  </HandlerName>
  <jxta:Cred>
    JXTACRED
  </jxta:Cred>
  <QueryID>
    0
  </QueryID>
  <HC>
    0
  </HC>
  <SrcPeerID>
    urn:jxta:uuid-
59616261646162614A7874615032503304BD268FA4764960AB93A53D7F15044503
  </SrcPeerID>
  <Query>
    <!-- Query omitida para abreviar -->
  </Query>
</jxta:ResolverQuery>
```

Figura 12: Exemplo de uma Query do PRP

O PRP não garante que seja gerado qualquer resultado em resposta a uma *query*, mas como seria de esperar define o formato das mensagens a serem utilizadas para devolver respostas. Como acontece no caso da *query* estas são normalmente mensagens de protocolos de nível superior encapsuladas dentro de uma Response do PRP.

```
<xs:element name="ResolverResponse" type="ResolverResponse"/>
<xs:complexType name="ResolverResponse">
  <xs:sequence>
    <xs:element ref="jxta:Cred" minOccurs="0"/>
    <xs:element name="ResPeerID" type="jxta:JXTAID" minOccurs="0" />
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string"/>
    <xs:element name="Response" type="xs:anyType"/>
  </xs:sequence>
</xs:complexType>
```

Figura 13: XML Schema de uma *Response* do PRP

Significado de cada elemento:

- `<jxta:Cred>` – Uma credencial de identificação do peer.
- `<ResPeerID>` – A identificação do *peer* que enviou a *Response*
- `<HandlerName>` – Uma string utilizada para decidir como e por quem deve ser interpretado o que aparecer contido no elemento *Response*.
- `<QueryID>` – Este elemento transita da mensagem de *query*, para ser possível ligar esta resposta a *query* que a gerou.
- `<Response>` – A resposta a *query* feita. Pode ser uma *string* ou uma outra mensagem encapsulada.

Na Figura 14 pode ver-se o aspecto que uma *Response* do PRP tem. O conteúdo do elemento `<Response>` foi omitido para abreviar a representação.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:ResolverResponse>
<jxta:ResolverResponse xmlns:jxta="http://jxta.org">
  <HandlerName>
    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000305
  </HandlerName>
  <jxta:Cred>
    JXTACRED
  </jxta:Cred>
  <QueryID>
    0
  </QueryID>
  <ResPeerID>
   urn:jxta:uuid-59616261646162614A7874615032503304BD268FA4764960AB93A53
D7F15044503
  </ResPeerID>
  <Response>
          <!-- Response omitida para abreviar -->
  </Response>
</jxta:ResolverResponse>
```

Figura 14: Exemplo de uma Response do PRP

3.7.4 Pipe Binding Protocol (PBP)

Este protocolo é utilizado para definir um *pipe*, e como deve ser usado esse *pipe*. Um *pipe* é um canal virtual que é estabelecido de forma a permitir aos *peers* comunicarem entre si. Este protocolo é construído por cima do PRP que utilizará o PEP caso exista uma rota directa entre os *peers* que necessitam de comunicar, ou o RVP caso a mensagem tenha de ser propagada até chegar ao destino.
Um *pipe* no JXTA é uma construção análoga ao *pipe* encontrado nos sistemas UNIX, em que a informação colocada do lado emissor viaja pelo *pipe* até chegar ao lado do receptor, formando assim um canal de comunicação unidireccional. Existem neste momento três tipos de *pipes* definidos pela especificação do JXTA,sendo que mais tipos podem ser construídos usando como base estes três tipos

- *Unicast* – Este é o tipo de *pipe* mais comum. Fornece um canal de comunicação unidireccional sem qualquer garantia a nível de fiabilidade ou a nível de segurança entre dois *peers*.

- *Secure Unicast* – Este tipo é exactamente igual ao anterior, com a diferença que estabelece uma ligação TLS entre os dois *peers* para proteger os dados que estejam a circular no *pipe*.

  Propagate – Este tipo é um derivado do tipo *Unicast*, com a diferença que uma mensagem enviada por um output pipe deste tipo é entregue (propagada) a todos os *peers* que tenham um input pipe correspondente, criando assim comunicação um-para-muitos, ao invés da um-para-um dos outros dois tipos.

Para descrever um *pipe* é utilizado um *advertisement*, que descreve todas as características necessárias para o pipe ser utilizado. A Figura 15 mostra o *schema* de um *Pipe Advertisement*:

```
<xs:element name="PipeAdvertisment" type="jxta:PipeAdvertisment"/>

<xs:complexType name="PipeAdvertisement">
  <xs:sequence>
    <xs:element name="Id" type="jxta:JXTAID" />
    <xs:element name="Type" type="xs:string" />
    <xs:element name="Name" type="xs:string" minOccurs="0" />
    <xs:element name="Desc" type="xs:anyType" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Figura 15: XML *Schema* de um *Pipe Advertisement*

Significado de cada elemento:

- `<Id>` – A identificação do pipe.
- `<Type>` – O tipo do *pipe*, de acordo com o definido nas especificações. Para um pipe do tipo *Unicast* o valor deve ser `JxtaUnicast`, para um pipe do tipo *Secure Unicast* o valor deve ser `JxtaUnicastSecure` e para um pipe do tipo *Propagate* o valor deve ser `JxtaPropagate`
- `<Name>` – Elemento opcional que permite associar um nome a um pipe. Os nomes não são necessariamente únicos ou seja múltiplos *peers* podem dar o mesmo nome aos seus *pipes*.
- `<Desc>` – Elemento opcional que permite associar uma descrição ou outras informações ao *advertisement*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
  <Id>urn:jxta:uuid-094AB61B99C14AB694D5BFD56C66E512FF7980EA1E6F4C238A26B
B362B34D1F104</Id>
  <Type>JxtaUnicast</Type>
  <Name>Talk to Me!</Name>
</jxta:PipeAdvertisement>
```

Figura 16: Exemplo de um *Pipe Advertisement*

Para a comunicação entre *peers* ser estabelecida via pipe, em primeiro lugar o *peer* que vai receber mensagens deve criar um pipe e um *advertisement*. Depois os *peers* interessados em utilizar o pipe devem descobrir o *advertisement*, por exemplo utilizando os mecanismos de descoberta automática da rede e quando na posse do *advertisement* criam um pipe que se vai ligar ao pipe lá descrito. Para localizar o *peer* que criou o pipe e o *advertisement* (a existência do *advertisement* não quer dizer necessariamente que quem o criou ainda se encontra na rede ou que o pipe ao qual se refere ainda se encontra disponível) é enviada então uma *query* numa mensagem do tipo *Pipe Resolver Message*, a qual se deve seguir uma resposta para o ínicio da comunicação, ou nada em caso de erro. Tanto a *query* como resposta utilizam o s*chema* da Figura 17.

```
<xs:element name="PipeResolver" type="jxta:PipeResolver"/>
<xs:simpleType name="PipeResolverMsgType">
  <xs:restriction base="xs:string">
    <!-- QUERY -->
    <xs:enumeration value="Query"/>
    <!-- ANSWER -->
    <xs:enumeration value="Answer"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="PipeResolver">
  <xs:sequence>
    <xs:element name="MsgType" type="jxta:PipeResolverMsgType"/>
    <xs:element name="PipeId" type="jxta:JXTAID"/>
    <xs:element name="Type" type="xs:string"/>
    <!-- used in the query -->
    <xs:element name="Cached" minOccurs="0" default="true"
type="xs:boolean"/>
    <xs:element name="Peer" minOccurs="0" maxOccurs="unbounded"
type="jxta:JXTAID" />
    <!-- used in the answer -->
    <xs:element name="Found" minOccurs="0" type="xs:boolean"/>
    <!-- This should refer to a peer adv, but is instead a whole doc -->
    <xs:element name="PeerAdv" minOccurs="0" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Figura 17: XML Schema de uma Pipe Resolver Message

Significado de cada elemento:

- `<MsgType>` – Este elemento indica qual o tipo de mensagem, podendo ter o valor de Query ou Answer
- `<PipeId>` – A identificação do pipe que está a ser localizado.
- `<Type>` – O tipo de pipe que está a ser localizado. O valor deste elemento deve coincidir com o valor do elemento <Type> do *pipe advertisement*.
- `<Cached>` – Utilizado para decidir se um *peer* que não possua o *pipe* deveria reenviar a *query* para os *peers* que considerasse que pudessem ter o *pipe*. Sem significado na versão actual do protocolo. O valor deste elemento deve ser definido sempre como falso.
- `<Peer>` – Se a mensagem for uma *query* este elemento representa o identificador do único *peer* do qual são esperadas respostas. Se a mensagem for uma *answer*, deve conter os identificadores de todos os *peers* que possuam um pipe correspondente.
- `<Found>` – Este elemento é usado para indicar se o pipe pretendido foi encontrado no *peer*.
- `<PeerAdv>` – O *advertisement* que representa o *peer* que possui o pipe pretendido.

De forma a controlar a propagação de mensagens pela rede quando se usa um pipe do tipo *Propagate* é adicionado a cada mensagem um cabeçalho sob a forma de um elemento cujo *schema* pode ser visto na Figura 18. Este cabeçalho é colocado no *namespace* reservado para utilização do JXTA.

```
<xs:element name="JxtaWire" type="jxta:JxtaWire"/>
<xs:complexType name="jxta:JxtaWire">
  <xs:sequence>
    <xs:element name="SrcPeer" minOccurs="0" type="jxta:JXTAID" />
    <xs:element name="PipeId" type="jxta:JXTAID" />
    <xs:element name="MsgId" type="xs:string" />
    <xs:element name="TTL" type="xs:unsignedInt" />
    <xs:element name="VisitedPeer" type="jxta:JXTAID"
maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Figura 18: XML *Schema* do cabeçalho das mensagens enviadas por um *Propagate Pipe*

Significado de cada elemento:

- `<SrcPeer>` – Este elemento representa o identificador do *peer* que enviou originalmente a mensagem.
- `<PipeId>` – Este elemento representa o identificador do *pipe* pela qual a mensagem está a ser enviada.
- `<MsgId>` – Este elemento representa o identificador da mensagem que está a ser enviada. É utilizado para detectar mensagens duplicadas a circular na rede.
- `<TTL>` – Este elemento é utilizado para controlar a propagação da mensagem na rede. Cada que *peer* que reenvie a mensagem deve subtrair uma unidade ao valor deste elemento, para que quando o valor chegue a zero a mensagem deixe de ser propagada.
- `<VisitedPeer>` – Este elemento contêm uma lista de identificadores de *peers* que já receberam esta mensagem. Ao reenviar a mensagem um *peer* deve acrescentar o seu identificador ao final desta lista, e fazer o que for possível para não propagar a mensagem para *peers* que se encontrem nesta lista.

3.7.5 Peer Information Protocol (PIP)

Este protocolo é utilizado para obter informações sobre outro *peer*.

É um protocolo cuja implementação é completamente opcional, o que implica que não existem quaisquer garantias de que um *peer* vá responder a mensagens deste protocolo, nem existem garantias sobre o conteúdo dessas mensagens em caso de resposta. Este protocolo funciona enviando as suas mensagens nos elementos <Query>/<Response> presentes nas mensagens do PRP.

```
<xs:element name="PeerInfoQueryMessage" type="jxta:PeerInfoQueryMessage"/
>

<xs:complexType name="PeerInfoQueryMessage">
  <xs:sequence>
    <xs:element name="sourcePid" type="jxta:JXTAID" />
    <xs:element name="targetPid" type="jxta:JXTAID" />
    <!-- if not present then the response is the general peerinfo -->
    <xs:element name="request" type="xs:anyType" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Figura 19: XML Schema de uma Query do PIP

Significado de cada elemento:

- `<sourcePid>` – A identificação do *peer* que enviou a *Query*.
- `<targetPid>` – A identificação do *peer* para a qual a *Query* é enviada.
- `<request>` – Elemento opcional que define o tipo de *Query*. O protocolo não define a estrutura deste elemento de forma a possibilitar a sua utilização por protocolos definidos pelo utilizador.

A Figura 20 mostra o XML schema utilizado nas Responses enviadas pelo PIP.

```
<xs:element name="PeerInfoResponseMessage"
type="jxta:PeerInfoResponseMessage"/>

<xs:complexType name="PeerInfoResponseMessage">
  <xs:sequence>
    <xs:element name="sourcePid" type="jxta:JXTAID" />
    <xs:element name="targetPid" type="jxta:JXTAID" />
    <xs:element name="uptime" type="xs:unsignedLong" minOccurs="0" />
    <xs:element name="timestamp" type="xs:unsignedLong" minOccurs="0" />
    <xs:element name="response" type="xs:anyType" minOccurs="0" />
    <xs:element name="traffic" type="jxta:piptraffic" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="piptraffic">
  <xs:sequence>
    <xs:element name="lastIncomingMessageAt" type="xs:unsignedLong"
minOccurs="0" />
    <xs:element name="lastOutgoingMessageAt" type="xs:unsignedLong"
minOccurs="0" />
    <xs:element name="in" type="jxta:piptrafficinfo" minOccurs="0" />
    <xs:element name="out" type="jxta:piptrafficinfo" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="piptrafficinfo">
  <xs:sequence>
    <xs:element name="transport" maxOccurs="unbounded">
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base="xs:unsignedLong">
            <xs:attribute name="Expiration" type="xs:anyURI" />
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figura 20: XML Schema de uma Response do PIP

Significado de cada elemento:

- `<sourcePid>` – A identificação do *peer* que enviou a *Query*.
- `<targetPid>` – A identificação do *peer* que recebeu a *Query*.
- `<uptime>` – Este elemento representa o número de milissegundos desde que o serviço do PIP começou a correr.
- `<timestamp>` – A identificação do *peer* que recebeu a *Query*.
- `<response>` – Este elemento pode conter a resposta a um *Request* feito na *Query* que despoleta a *Response*. Para se conseguir efectuar uma correspondência entre *Responses* e *Queries* o valor do elemento `<QueryID>` presente na mensagem do PRP que é utilizada para encapsular as mensagens do PIP deve ser igual tanto na mensagem que originou a *Query* como nas *Responses* que são geradas.
- `<trafic>` – Elemento opcional que fornece informações sobre o tráfego que o *peer* recebe e envia.
- `<lastIncomingMessageAt>` – Este elemento representa o último instante em que o *peer* recebeu uma mensagem válida. É representado em milissegundos e medido em relação a 1 de Janeiro de 1970, 00:00:00 GMT
- `<lastOutgoingMessageAt>` – Este elemento representa o último instante em que o *peer* enviou uma mensagem válida. É representado da mesma forma que o elemento anterior.
- `<in>` – Este elemento contem informações sobre o tráfego recebido nos vários *endpoints* disponíveis.
- `<out>` – Este elemento contem informações sobre o tráfego enviado pelos vários *endpoints* disponíveis.
- `<transport>` – Quando presente num elemento `<in>` representa o número de bytes recebidos num *endpoint*, quando se encontra num elemento `<out>` representa o número de bytes enviados por um *endpoint*.

3.7.6 Peer Discovery Protocol (PDP)

Este protocolo é utilizado para a descoberta de recursos anunciados na rede.

Os recursos são representados por *advertisements* XML, e podem representar tudo desde outro *peer* a um serviço disponível. Este protocolo funciona de duas formas distintas: numa rede local utiliza mensagens *multicast* de forma a descobrir recursos (pesquisa directa, Figura 21), ou então pode direccionar os seus pedidos para um *peer* do tipo *Rendezvous* (pesquisa indirecta) que pode possuir no seu índice informação para responder ao pedido directamente ou pode reencaminhar os pedidos para outros *Rendezvous* (com um limite no número de reencaminhamentos possíveis de modo a evitar ciclos infinitos) no caso de não ser capaz de responder à *query* que lhe foi enviada.

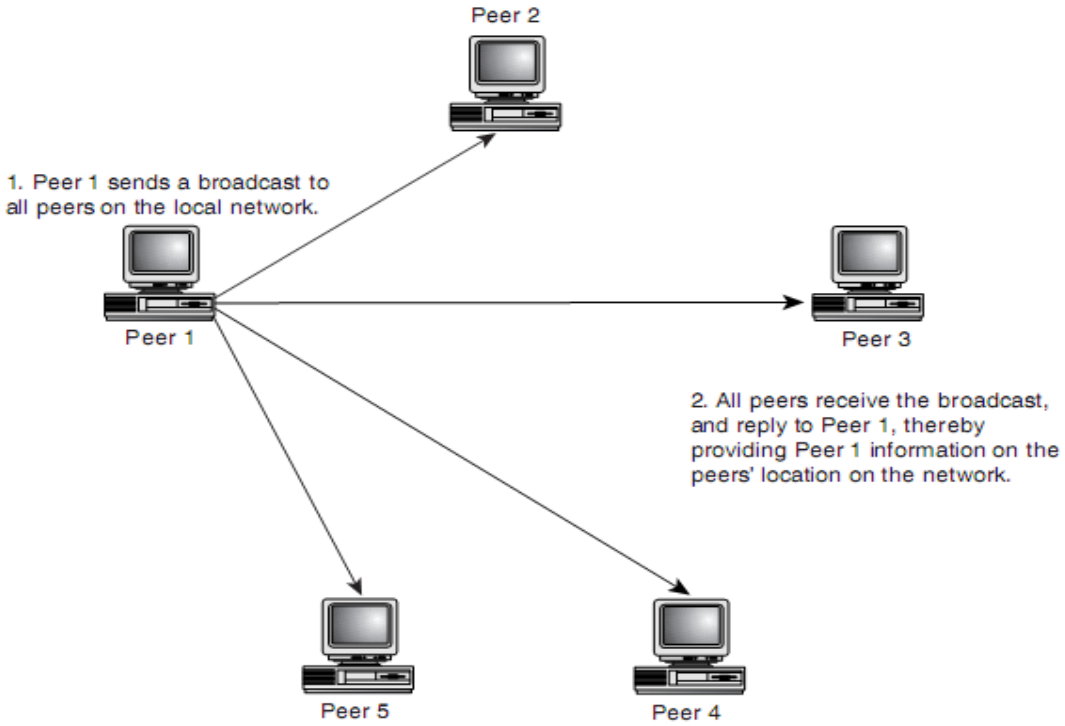

Figura 21: Pesquisa Directa [9]

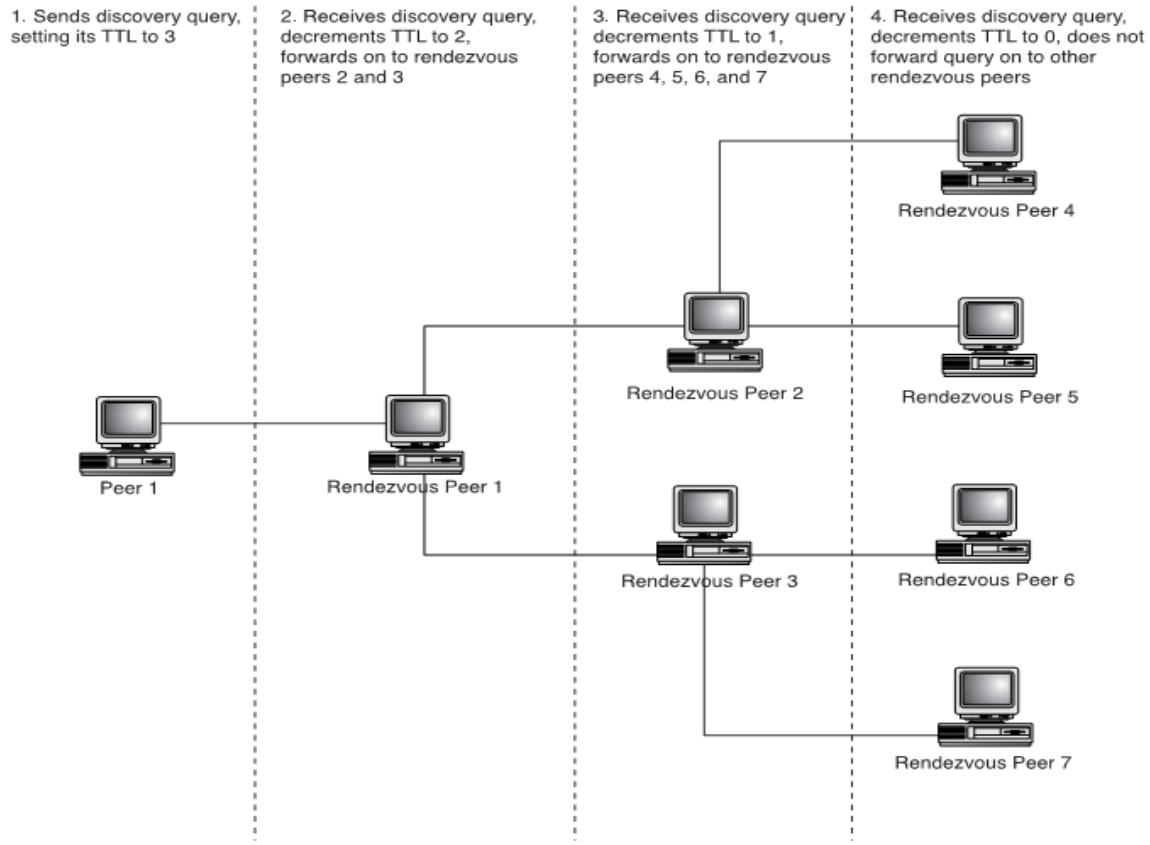

Figura 22: Pesquisa Indirecta [9]

Para tornar a descoberta de recursos mais eficiente os *peers* implementam um mecanismo de cache dos *advertisements* que vão encontrando. Assim antes de procurar na rede um *peer* pode escolher consultar a sua cache local (correndo o risco de possuir um *advertisement* para algo que no momento já não exista). Quando é efectuada uma pesquisa pela rede os *advertisements* recebidos são armazenados na cache local para futura utilização. Esta estratégia de pesquisa pode ser vista na Figura 23.

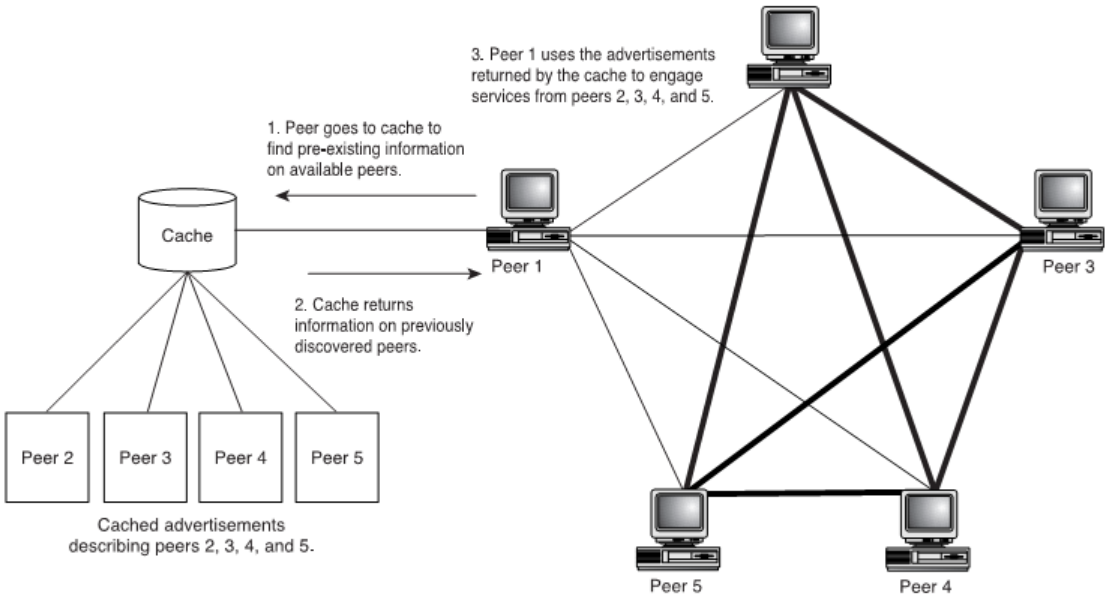

Figura 23: Utilização da cache de *advertisements* do JXTA [9]

As *queries* (Figura 24) e *responses* (Figura 25) definidas por este protocolo são enviadas recorrendo ao PRP.

```
<xs:element name="DiscoveryQuery" type="jxta:DiscoveryQuery"/>
<xs:complexType name="DiscoveryQuery">
  <!-- this should be an enumeration -->
  <xs:element name="Type" type="xs:string"/>
  <xs:element name="Threshold" type=”xs:unsignedInt" minOccurs="0"/>
  <xs:element name="PeerAdv” type="jxta:PA" minOccurs="0"/>
  <xs:element name="Attr" type="xs:string" minOccurs="0"/>
  <xs:element name="Value" type="xs:string" minOccurs="0"/>
</xs:complexType>
```

Figura 24: XML *Schema* de uma *Query* do PDP

Significado de cada elemento:

`<Type>` – O tipo de *advertisement* que está a ser pesquisado. Existem três tipos de valores possíveis para este elemento :

- PEER – Este valor indica que devem ser pesquisados *advertisements* com informações sobre *peers* individuais.
- GROUP – Este valor indica que devem ser pesquisados *advertisements* com informações sobre um *peer group*

- ADV – Este valor indica que não existem restrições ao tipo de *advertisements* que devem ser pesquisados

- `<Threshold>` – O número máximo de respostas que devem ser devolvidas ao *peer* que fez a pesquisa por cada *peer* que lhe envie uma *response*.
- `<PeerAdv>` – O *advertisement* do *peer* que fez a pesquisa.
- `<Attr>` – Um elemento especifico dos *advertisements* que devem ser retornados e sobre o qual é efectuada uma filtragem com base no valor do elemento `<Value>`.
- `<Value>` – O valor utilizado para filtrar os *advertisements*.

```
<xs:element name="DiscoveryResponse" type="jxta:DiscoveryResponse"/>
<xs:complexType name="DiscoveryResponse">
  <!-- this should be an enumeration -->
  <xs:element name="Type" type="xs:string"/>
  <xs:element name="Count" type="xs:unsignedInt" minOccurs="0"/>
  <xs:element name="PeerAdv" type="xs:anyType" minOccurs="0">
    <xs:attribute name="Expiration" type="xs:unsignedLong"/>
  </xs:element>
  <xs:element name="Attr" type=”xs:string" minOccurs="0"/>
  <xs:element name="Value" type=”xs:string" minOccurs="0"/>
  <xs:element name="Response" type="xs:anyType" maxOccurs="unbounded">
    <xs:attribute name=vExpiration" type="xs:unsignedLong"/>
  </xs:element>
</xs:complexType>
```

Figura 25: XML Schema de uma Response do PDP

Significado de cada elemento:

- `<Type>` – O tipo de *advertisement* que é devolvido em cada elemento `<Response>`.
- `<Count>` – O número total de respostas obtidas. Neste caso vai definir quantos elementos `<Response>` existem nesta mensagem.
- `<PeerAdv>` – O *advertisement* do *peer* que está a responder a *query*. Possui um atributo que explicita o tempo em milissegundos até que *advertisement* deixa de ser válido. Quando o tempo indicado expirar o *advertisement* deve ser removido da cache do *peer*..
- `<Attr>` – O atributo que foi utilizado para pesquisar os *advertisements* que são devolvidos nesta mensagem.

- `<Value>` – O valor do atributo que foi utilizado para a pesquisa.
- `<Response>` – Este elemento contem os *advertisements* encontrados pela pesquisa. O número de resultados encontrados é indicado pelo valor do elemento `<Count>`. Cada elemento deste tipo tem um atributo que indica o tempo de vida do *advertisement* em milissegundos.

3.8 *JXTA e WebServices*

Uma grande parte do esforço de desenvolvimento actual é colocada em SOA (*Service Oriented Architecture)*. Este tipo de arquitecturas tem normalmente um modelo cliente-servidor cujos serviços estão disponíveis via servidor centralizado que se assume estar sempre disponível e com um endereço conhecido. Quando o serviço fornecido se torna popular o servidor que o disponibiliza pode tornar-se incapaz de lidar com todos os pedidos, o que leva a criação de *clusters* de servidores para lidar com esta situação. É neste caso que entra abordagem *peer-to-peer*. Se em vez de desenharmos uma topologia cliente-servidor pura desenhássemos uma rede de *peers* que poderiam fornecer serviços, poderíamos eliminar a necessidade de existir um endereço conhecido para um servidor, bastaria pesquisar na rede por um *peer* que fornecesse o serviço [18]. Isto levaria também à quebra da necessidade de um servidor em particular estar sempre disponível. Uma forma popular de implementar uma SOA é utilizar *Webservices*. Seria interessante continuar a utilizar a mesma abordagem mas com os benefícios acrescidos oferecidos pela abordagem peer-to-peer descrita.

Como já foi visto na secção 3.3 o JXTA possui uma abstracção nativa para a criação de funcionalidades adicionais, que promove a criação de implementações quase *ad-hoc* de serviços que podem ser usados pelos restantes *peers*. Sobre esta abstracção foi implementada uma biblioteca [19] que permite disponibilizar serviços na rede criada pelo JXTA usando a maioria das premissas dos *Web Services*, ou seja existe uma implementação do protocolo SOAP (*Simple Object Access Protocol*) [20] sobre o JXTA. Quando se quer disponibilizar um serviço desta forma é necessário criar um descritor WSDL (*Web Services Description Language*) [21] que descreve o serviço a fornecer. Quando o serviço é instanciado é criado um *Module Class Advertisement* e o respectivo *ModuleSpec*

Advertisement, no qual é colocado um *Pipe Advertisement* do *pipe* que será utilizado pelo serviço e no elemento `<Param>` é colocado o documento WSDL anteriormente criado. Depois os *advertisements* são difundidos pela rede para serem encontrados pelos clientes. O serviço em si corre num servidor Apache Axis [22], que permite que o protocolo de transporte das mensagens SOAP seja o JXTA em vez do tradicional HTTP. Os clientes devem procurar na rede um *advertisement* de um serviço que lhes interesse, recuperar o WSDL presente no *advertisement* e tratá-lo como se fosse um *webservice* normal (cuja criação do objecto que faz a chamada é gerida pela biblioteca JXTA-SOAP).

4 Arquitectura P2P para suportar serviços de uma Biblioteca Digital

Esta dissertação visa o estudo da possibilidade da aplicação de tecnologias *peer-to-peer* no âmbito das bibliotecas digitais. Para esse efeito foi escolhido como *framework peer-to-peer* o JXTA, descrito no capítulo anterior. Dado que JXTA possui implementações em várias linguagens de programação (*bindings*) foi necessário escolher qual desses *bindings* utilizar. A escolha foi limitada pela necessidade de fazer interacções do tipo *Web Services*. Dado que o projecto JXTA-SOAP neste momento só fornece uma implementação de referência em Java esta foi a linguagem de programação escolhida.

Os grandes objectivos deste projecto foram a integração de uma tecnologia de indexação baseada no Apache Lucene [23] com o JXTA, de forma a permitir a pesquisa distribuída sobre os índices gerados por vários *peers* e quando fosse caso disso a obtenção dos conteúdos referenciados e a criação de uma plataforma de suporte a serviços de suporte a bibliotecas digitas. Para simplificar a infra-estrutura necessária, assume-se por agora que a componente *peer-to-peer* vai funcionar isolada numa rede local sem contacto com o exterior.

4.1 *Arquitectura da rede*

Apesar de nesta fase se assumir que toda a infra-estrutura *peer-to-peer* se encontra contida numa única rede local é necessário reflectir um pouco sobre a arquitectura que a rede *peer-to-peer* deve adoptar.

Num cenário com poucos *peers* não há necessidade evidente para a criação de *super peers* e da adopção de uma topologia híbrida, o que faz com que a adopção de uma arquitectura descentralizada pareça uma opção viável. Neste tipo de cenário os *peers* iriam servir-se do mecanismo de *advertisements* do JXTA para encontrarem outros *peers*, fazendo pesquisas sobre os índices de cada *peer* de forma independente. A vantagem desta abordagem seria a de garantir que as pesquisas seriam sempre efectuadas sobre as versões mais recentes dos índices, sendo que a desvantagem seria a quantidade de tráfego gerada pelas pesquisas.

Por outro lado à medida que a rede vai crescendo (e eventualmente quando a limitação de estar contida numa rede local deixar de se colocar) torna-se claro que a adopção de uma arquitectura híbrida com a criação de um *super peer* (de notar que um *super peer* neste contexto não tem necessariamente de ser um *super peer* segundo a definição do JXTA), cujo papel seria o de agregar os índices de todos os *peers* da rede, iria contribuir para aumentar a fiabilidade da rede pois promove uma ligação entre *peers* que poderiam de outra forma não se conhecer, evitando uma possível perda de resultados nas pesquisas efectuadas na rede. Esta abordagem apresenta a vantagem de reduzir o tráfego na rede quando são efectuadas pesquisas dado que estas são dirigidas ao *super peer* e é apenas dele que vem as respostas. As desvantagens seriam que nem sempre se estaria a fazer pesquisas sobre a versão mais recente dos índices de um *peer* e a transferência do índice para o *super peer* e a respectiva actualização geram algum tráfego adicional.

De forma a aproveitar as vantagens de cada uma destas arquitecturas foi definida uma topologia híbrida opcional: um *peer* pode escolher ligar-se a um *super peer*, registar-se e transferir para lá o seu índice e fazer as suas pesquisas sobre os índices contidos no *super peer*, ou manter-se à margem e fazer as suas pesquisas directamente sobre os *peers* que conhecer. Em caso de falha do *super peer* os *peers* adoptam automaticamente este último comportamento.

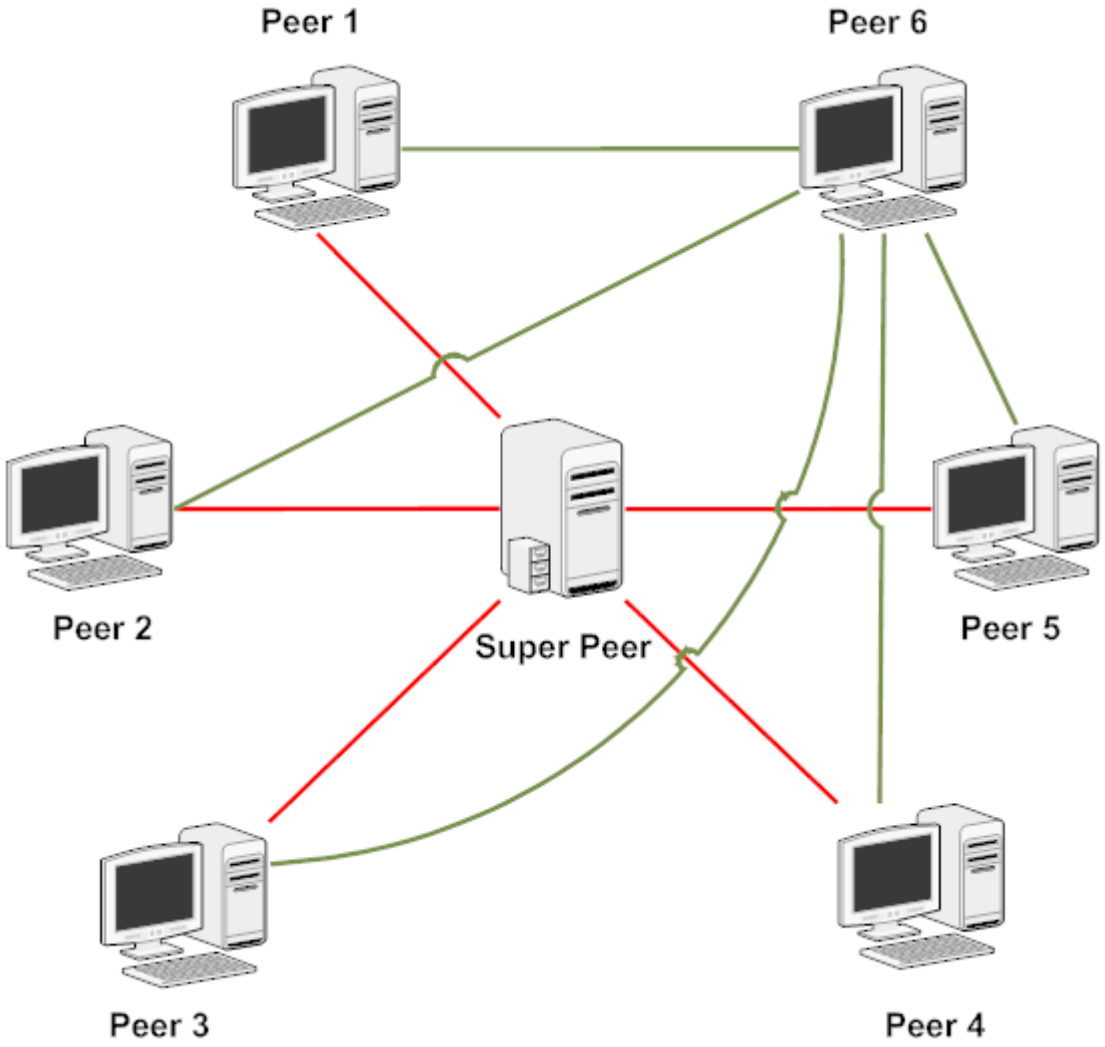

Figura 26: Arquitectura da rede ao utilizar um *super peer*

Na Figura 26 pode ver-se o comportamento da rede na presença de um *Super Peer*. Os *peers* 1-5 decidiram usar o *super peer* para auxiliar as suas pesquisas na rede. Ao pesquisarem na rede encontram os conteúdos dos *peers* 1-5, que lhes são indicados pelo *super peer*. Por outro lado o *peer* 6 escolheu não utilizar o *super peer*. Como tal tem de estabelecer ligações (representadas a verde) com todos os outros *peers* de forma a pesquisar os seus conteúdos. Neste cenário os *peers* 1-5 não tem acesso aos conteúdos do *peer* 6, mas as suas pesquisas geram muito menos tráfego.

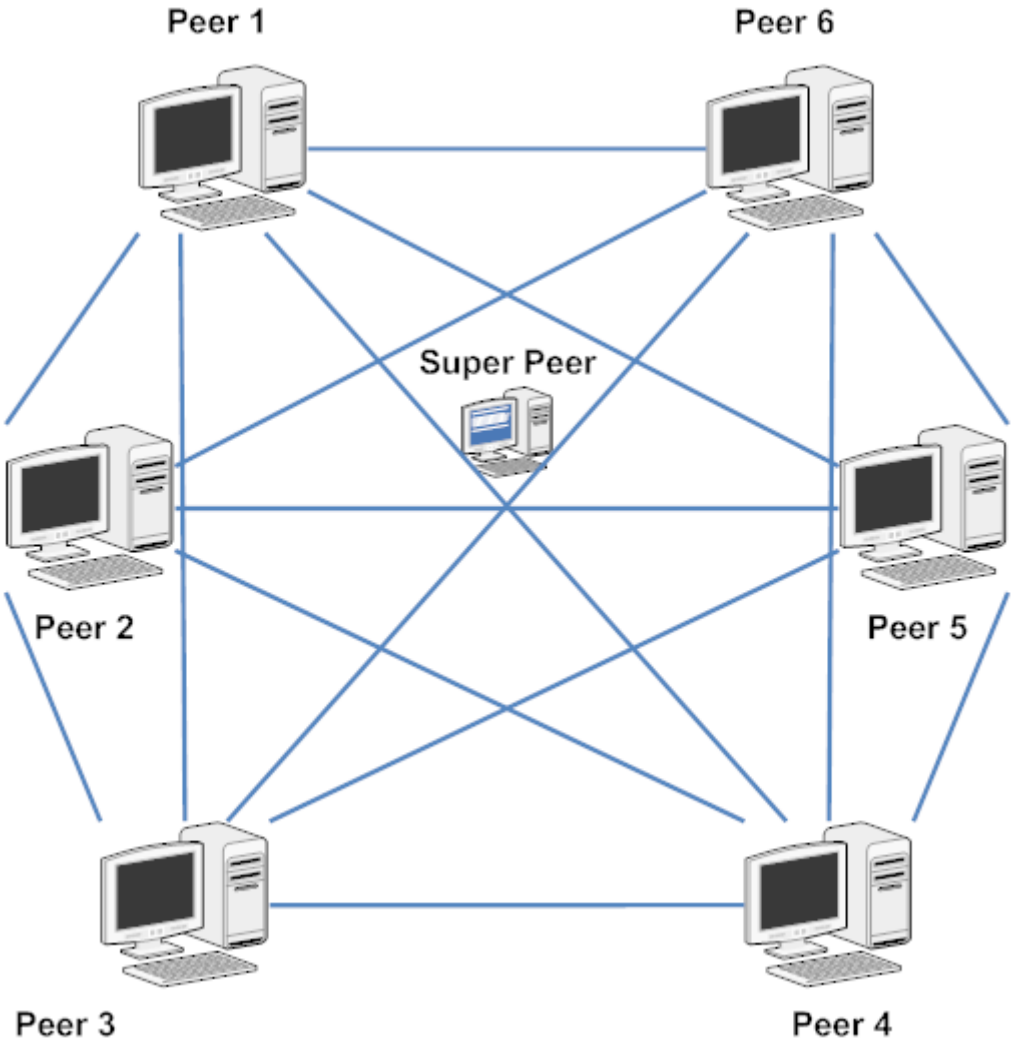

Figura 27: Arquitectura da rede sem um super peer

Na Figura 27 pode ver-se o comportamento da rede quando existe uma falha num *super peer*. Os *peers* 1-5 adoptam o comportamento que o *peer* 6 apresentava na Figura 26 e para efectuarem uma pesquisa devem estabelecer ligações com todos os outros *peers* que conheçam, garantido que a rede continua utilizável quando existe uma falha num *super peer* a custa de aumentar o tráfego gerado pelas pesquisas.

4.2 *Descoberta de peers*

De forma a tornar possível a arquitectura definida na secção anterior é necessário definir os valores que devem aparecer num *advertisement* de forma a uniformizar as pesquisas de *peers*. Dado que os *peers* devem possuir um pipe para

possibilitar a comunicação entre eles foi definido que seria utilizado um pipe *advertisement*. De forma a diferenciar estes *advertisements* dos restantes que possam existir na rede foi definido que o valor do elemento `<Name>` seria "JxtaIndexPeer". Para além disso no elemento `<Desc>` foi definido que seriam enviadas outras informações importantes para o funcionamento da rede. Foram definidos elementos básicos para cada informação necessária, e no futuro podem ser acrescentados mais elementos sem quebrar a compatibilidade com os *peers* já existentes.

Elementos definidos:

- `<PeerType>` – Utilizado para distinguir se um *peer* actua como *super peer*, caso no qual tem como valor "JxtaIndexServerPeer" ou como peer normal, caso no qual tem o valor "JxtaIndexPeer".
- `<PeerUUID>` – O identificador atribuído pelo JXTA a cada peer é utilizado principalmente para o encaminhamento de mensagens e varia entre *peer groups* e quando o *peer* é criado. Foi necessário criar um mecanismo que possa identificar um *peer* de forma única. Para tal foi escolhido gerar identificadores únicos universais (UUID) [24]. O valor deste elemento reflecte o identificador atribuído a cada *peer*.
- `<Workload>` – Este elemento foi definido para ser utilizado pelos serviços criados na rede. Espelha o número de pedidos feitos a um *peer*. Dado que os *advertisements* possuem um tempo de vida finito ao fim do qual deixam de ser validos tendo de ser republicados, este tempo é aproveitado para calcular os pedidos recebidos pelo *peer* no intervalo entre a publicação dos *advertisements*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
  <Id>urn:jxta:uuid-094AB61B99C14AB694D5BFD56C66E512FF7980EA1E6F4C238A26
BB362B34D1F104</Id>
  <Type>JxtaUnicast</Type>
  <Name>JxtaIndexPeer</Name>
  <Desc>
    <PeerType> JxtaIndexPeer </PeerType>
    <PeerUUID> d4d8bfc2-5afc-476c-a20d-eefa3ba23422 </PeerUUID>
    <Workload> 0 </Workload>
  </Desc>
</jxta:PipeAdvertisement>
```

Figura 28: Exemplo de um *pipe advertisement* utilizado pelos *peers*

4.3 *Canais de comunicação*

Na secção anterior é mencionado que os *peers* devem possuir um pipe para possibilitar a comunicação entre eles. Esta secção procura detalhar o tipo de pipe a ser utilizado. Já foi referido anteriormente que o JXTA oferece três tipos de pipes diferentes, que possuem uma característica comum: são unidireccionais. Esta característica faz com que o sistema de pipes seja uma construção de baixo nível cuja utilização não é recomendada [15] e para a qual é necessário oferecer alternativas. Para tal a implementação em Java do JXTA oferece duas abstracções de alto nível sobre os *pipes* unidireccionais, os BidiPipes (pipes bidireccionais) e os JxtaSockets. Os BiDipipes possuem uma interface orientada a transferência de mensagens (com um tamanho máximo de 64kb por mensagem que cabe ao programador utilizador respeitar) enquanto que os JxtaSockets possuem uma interface orientada a *streams* de informação. Ambas as abstracções fornecem um modo de funcionamento semelhante aos tradicionais ServerSocket/Sockets do Java, como se pode ver na Figura 29.

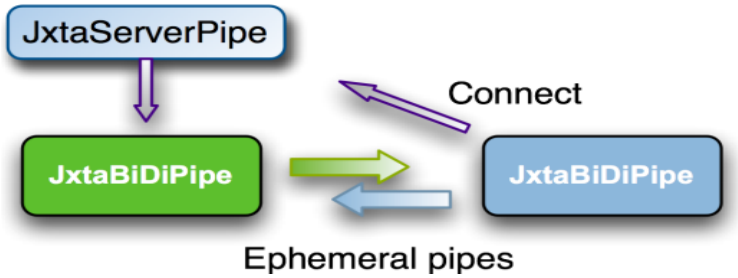

Figura 29: Funcionamento dos *pipes* Bidireccionais [15]

Para este projecto optou-se por utilizar os pipes bidireccionais, o que implica que cada *peer* deve criar um JxtaServerPipe, com um *advertisement* análogo ao visto na Figura 28.

Ao receber uma ligação o peer deve determinar o tipo da mensagem e criar um *handler* apropriado para lidar com a mensagem. Para que o *peer* possa identificar o tipo da mensagem, que as mensagens dirigidas a este pipe devem possuir um elemento de nome `<MessageType>` para que baseado nele se possa decidir como lidar com a mensagem. Todos os outros elementos devem ser interpretados apenas pelo *handler* definido para o tipo de mensagem. Se o tipo de mensagem não tiver um *handler* definido no peer este deve enviar uma resposta que contém apenas dois elementos: `<MessageType>` com o valor "Error" para indicar que existiu um erro e `<Reason>` cujo valor deve reflectir a natureza do erro e que fica a cargo do *peer* definir. Este formato de mensagem de erro é utilizado extensivamente por todos os *handlers*.

4.3.1 Message Handlers

A implementação do JXTA em Java fornece um mecanismo para definir uma forma de lidar com as mensagens que surgem num pipe. Cada pipe pode ter associado um *message listener*, um mecanismo de *callback* que recebe e processa as mensagens. Este mecanismo pode ser usado para dar diferentes tratamentos às mensagens. A escolha para canal de comunicação dos pipes bidireccionais impõe no entanto uma séria restrição ao uso deste mecanismo já que existe uma forte possibilidade de ocorrer um *deadlock* se for enviada uma mensagem da mesma classe designada para estar a processar as mensagens recebidas. De forma a contornar este problema foi criada uma classe específica para lidar com o atendimento de mensagens, que delega o processamento noutra classe (*message handler*) e garante que apenas uma mensagem é processada de cada vez. Uma terceira classe foi criada de forma a uniformizar o envio de mensagens.

O primeiro *message handler* é comum a todos os tipos de mensagens, e tem a responsabilidade de criar outro *message handler*, esse sim específico para cada tipo de mensagem. De forma a criar um sistema extensível, mas que possa ser

mantido com pouco esforço faz-se uso das capacidades de *reflection* do Java para criar os *message handlers* apropriados. Para esse efeito utiliza-se um ficheiro de propriedades, um simples ficheiro de texto que contém pares de valores separados pelo sinal de igual. Este ficheiro contém o tipo de mensagem, e a classe a ser invocada para fazer o respectivo processamento. Um exemplo de uma linha desse ficheiro seria algo como:

```
Keepalive = pt.ua.JxtaIndexPeer.MessageHandlers.Server.KeepAliveHandler
```

Com este mecanismo não é necessário alterar o código do primeiro *message handler* que é chamado quando for definido um novo tipo de mensagem, bastando apenas adicionar uma nova entrada no ficheiro de propriedades adequado.

Depois de criado um *message handler* processaria a mensagem de forma a decidir se a deve aceitar. Se aceitar deve informar o *listener* definido que passará a lidar com as mensagens recebidas naquele pipe, caso contrário deve fechar o pipe. Este processo é ilustrado pela Figura 30.

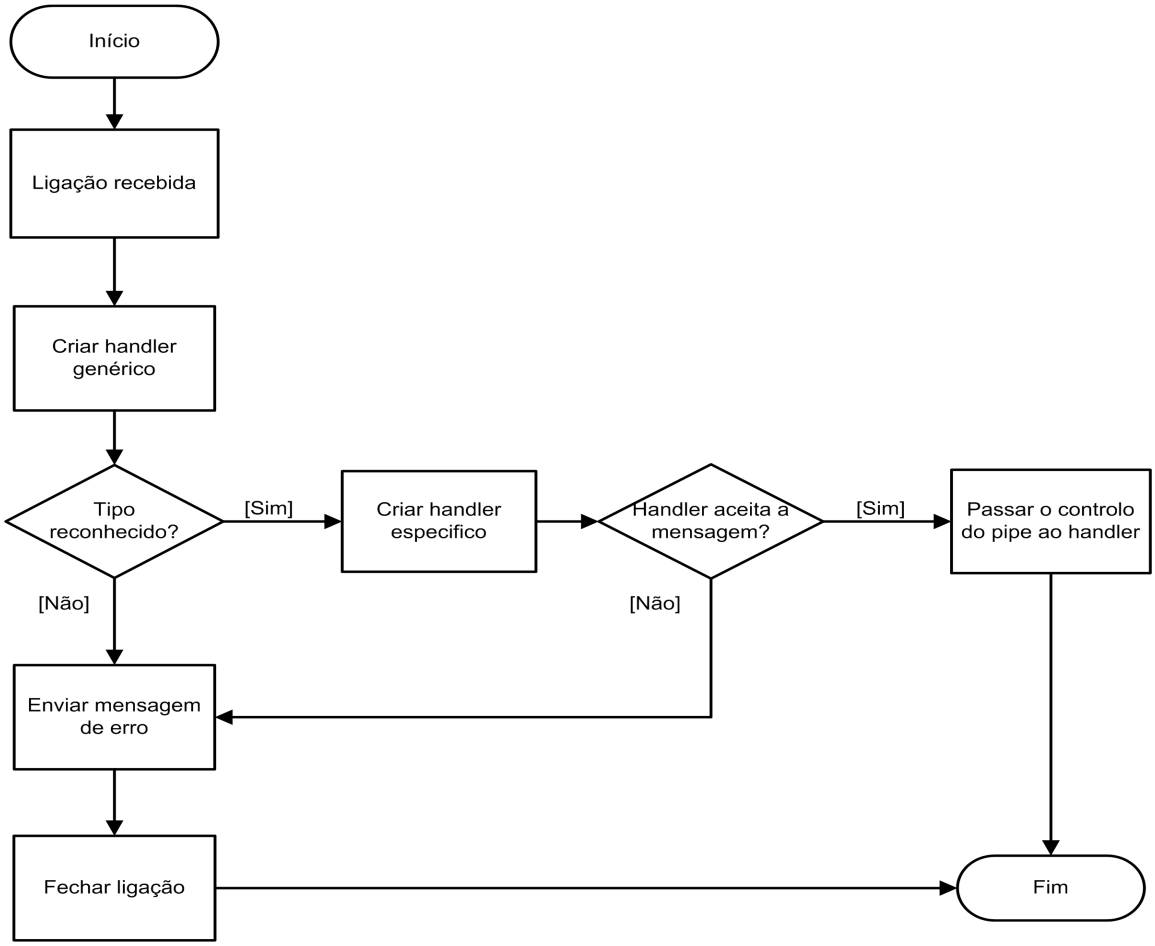

Figura 30: Processo de criação de *message handlers* específicos

4.4 *Registo com um super peer e transferência de índices*

Quando um *peer* decide que deve utilizar a infra-estrutura de *super peers* para auxiliar as suas pesquisas é necessário primeiro descobrir um *super peer*, pesquisando nos *pipe advertisements* que consiga encontrar. Assumindo que consegue encontrar um *pipe advertisement* de um *super peer* deve depois tentar registar-se, tarefa para a qual foi definido um tipo de mensagem. Para além do tipo são definidos alguns elementos que o *message handler* criado para lidar com o registo espera encontrar:

- `<MessageType>` – Este elemento define o tipo da mensagem. Neste caso o valor deve ser "RegisterRequest".
- `<UUID>` – Este elemento contem o valor do identificador único gerado pelo *peer*.
- `<Contact>` – Este elemento contem o identificador do pipe que o *peer* disponibiliza. Neste momento apesar de armazenado pelo *super peer* não é utilizado.
- `<IndexVersion>` – Este elemento reflecte a versão do índice do *peer*. Cada alteração ao índice faz mudar a versão.

Quando recebe uma mensagem com estas características o *super peer* processa a mensagem e tenta adicionar o *peer* a sua lista de *peers* conhecidos. Para um *super peer* auxiliar os *peers* nas suas pesquisas necessita de possuir os seus índices. No caso de ser a primeira vez que o *peer* se regista é criada uma pasta cujo nome é o UUID do *peer* para conter o seu índice e o *peer* é contactado depois de lhe ser enviada uma mensagem de confirmação de registo para transferir o seu índice para o *super peer*. Os *peers* podem escolher ignorar este pedido, e transferir o índice apenas quando for lhes conveniente, no entanto o comportamento implementado é o de transferir o índice a pedido do *super peer*. No caso de não ser a primeira vez que o *peer* se regista é utilizado o elemento `<IndexVersion>` para determinar se a versão do índice é ainda a mesma que o *super peer* possui. Se as versões diferirem é pedido ao *peer* uma actualização do

índice depois de lhe ser enviada uma mensagem a confirmar o registo. Desta forma evita-se transferir o índice todo novamente, uma operação que pode ser custosa se o índice for grande. O *super peer* implementado não espera nenhuma resposta directa aos pedidos de transferência de índice, e fecha imediatamente o pipe no fim de enviar a mensagem. As mensagens de confirmação do servidor seguem sempre o mesmo formato e possuem os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem. Neste caso o valor deve ser “ServerMessage”.
- `<Operation Result>` – Este elemento reflecte o resultado da operação. O valor dever ser “OK” em caso de sucesso.
- `<ServerID>` – Este elemento contem o identificador único do *super peer*. Os *peers* utilizam este valor para identificar novamente o *advertisement* do *super peer*. Desta forma podem eventualmente existir vários *super peers* na rede.

As mensagens de pedido de índice possuem apenas o elemento `<MessageType>`, cujo valor deve ser “IndexRequest” se o pedido for para uma transferência completa do índice ou “IndexUpdate” se o pedido é para uma actualização do índice.
Para transferir o índice para o *super peer*, um *peer* tem de iniciar um processo de negociação que começa com uma mensagem com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e também o tipo de transferência pretendido. Deve ter o valor “IndexTransfer” se for a transferência de um índice completo e o valor “IndexUpdate” se for a actualização do índice já existente.
- `<UUID>` – Este elemento contem o valor do identificador único gerado pelo *peer*.
- `<IndexVersion>` – Este elemento reflecte a versão do índice do *peer*

Se o *super peer* aceitar a transferência deve enviar uma mensagem de confirmação, após a qual o *peer* deve começar a transferência do índice. Os índices são compostos por múltiplos ficheiros. Para evitar múltiplas transferências individuais de cada ficheiros estes são comprimidos num arquivo *zip* e é calculado o *checksum* que este deve apresentar quando descomprimido. O arquivo a ser transferido é então dividido em blocos com no máximo 4kb.
Para a transferência do índice foi definido que a mensagem deve ter os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “IndexBlockTransfer”
- `<FileName>` – Este elemento indica o nome do ficheiro que está a ser transferido.
- `<Checksum>` – Este elemento guarda o valor do *checksum* que o ficheiro quando descomprimido deve apresentar. Se no final da transferência e descompressão o *checksum* for diferente do apresentado neste campo a transferência falhou.
- `<Block>` – Este elemento indica o bloco que está a ser transferido.
- `<Total Blocks>` – Este elemento indica o número total de blocos a ser transferido. Quando o valor do elemento `<Block>` for igual ao deste elemento o índice deve estar completamente no servidor.
- `<ByteCount>` – Este elemento indica quantos bytes possui o bloco de dados a ser transferido.
- `<Datablock>` – Este é o elemento onde os dados a ser transferidos são realmente colocados. Ao contrário de todos os outros elementos que eram do tipo *String* este é do tipo *ByteArray*.

O *super peer* ao receber uma destas mensagens deve confirmar a correcta recepção e escrita em disco com uma mensagem com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “Index Transfer Info”

- `<Status>` – Este elemento indica o estado da transferência do último bloco recebido. Deve apresentar o valor “OK” no caso de o bloco ter sido recebido com sucesso, o valor “Finished” no caso de o bloco recebido ser o último e a transferência ter terminado sem erros e “Failed” no caso de ter sido detectado um erro. Quando são detectados 10 erros numa única transferência esta é imediatamente abortada.

Uma possível sequência de mensagens resultante do processo de registo e transferência de índices pode ser visto na Figura 31.

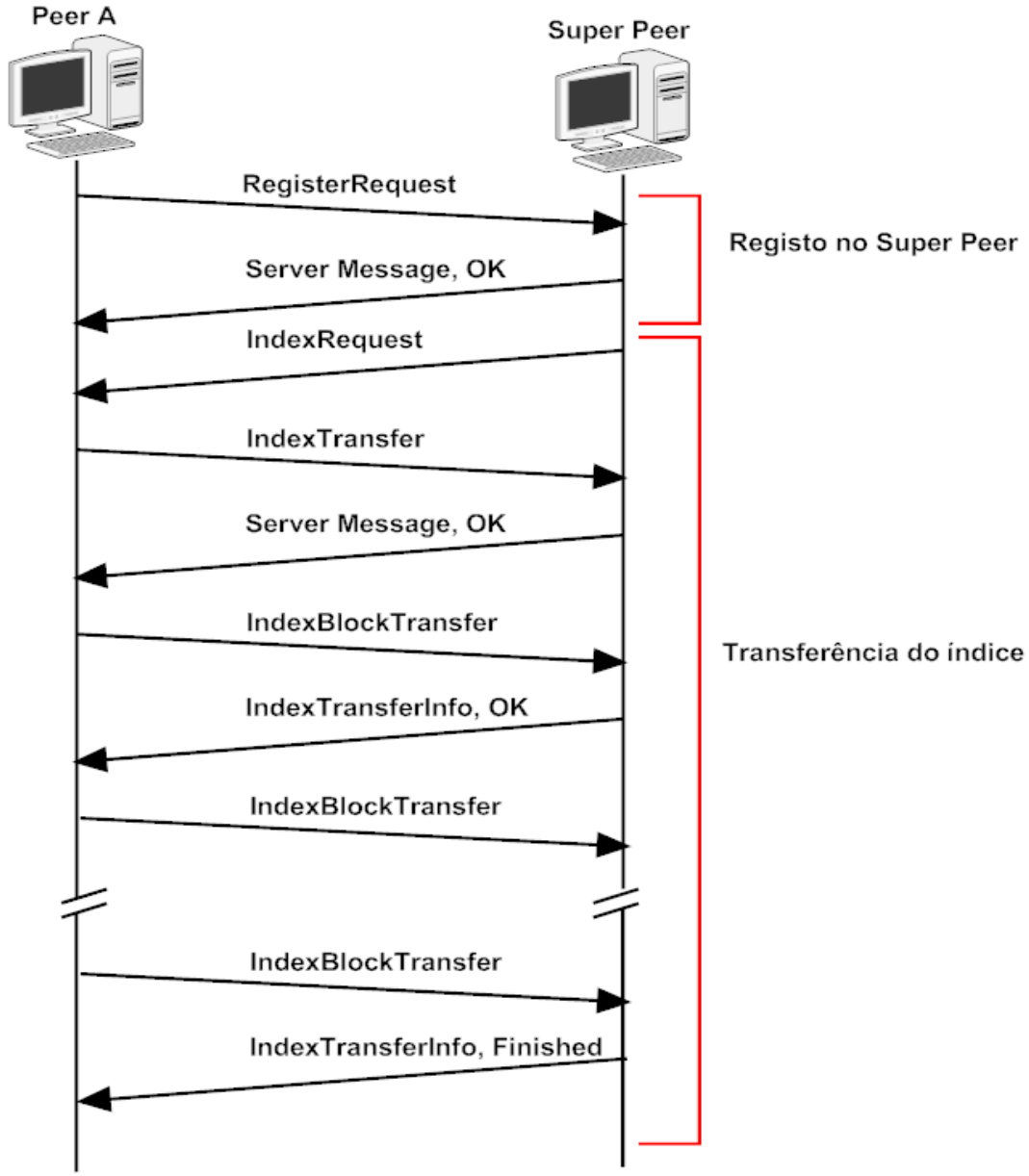

Figura 31: Sequência de mensagens de um registo e transferência de índice

4.4.1 Formato dos índices

Os índices dos *peers* são gerados pelo Apache Lucene e como tal os seus campos podem variar consoante o tipo de material que foi indexado. Os *peers* são livres de indexarem o que considerarem apropriado (o que se traduz numa potencial diversidade de conteúdos indexados) com os campos que considerarem mais apropriados para o conteúdo, desde que para cada entrada no índice indexem também os seguintes campos:

- `PeerUUID` – Este campo indexa o identificador do *peer* que originalmente adicionou esta entrada ao seu índice. É utilizado depois de uma pesquisa para identificar o *peer* ao qual terá de ser endereçado um pedido de transferência de ficheiro.
- `FileHash` – Este campo indexa um *hash* único utilizado para identificar o ficheiro que foi indexado. A combinação deste campo com o `PeerUUID` identifica um ficheiro para uma transferência.
- `FilePath` – Este campo indexa a localização do ficheiro indexado. Quando surge um pedido para uma transferência de um ficheiro o *peer* procura no seu índice pelo campo `FileHash` em busca de um resultado, de onde extrai o valor deste campo o que lhe permite iniciar a transferência.
- `FileName` – Este campo é indexado por conveniência. Se numa pesquisa o *peer* não definir os campos sobre os quais deve incidir a pesquisa é utilizado este campo para emular o comportamento dos tradicionais sistemas *peer-to-peer* de partilha de ficheiros em que as pesquisas são efectuadas pelo nome do ficheiro desejado.

4.4.2 Indexação de conteúdo

Como foi dito anteriormente os *peers* são livres de indexarem tudo o que considerem apropriado. De forma a suportar esta diversidade foi adoptado um esquema semelhante ao utilizado para gerir os *message handlers*, recorrendo a

um ficheiro de propriedades que associa uma extensão de um ficheiro à classe que deve criar uma entrada que possa ser adicionada ao índice.
Foram criadas classes que indexam ficheiros de texto (com a extensão txt e xml), documentos do tipo *Portable Document Format* (com a extensão pdf) e ficheiros de áudio (com a extensão mp3). Os tipos de ficheiros indexados demonstram a versatilidade de conteúdos que podem ser indexados. Como regra geral desde que os ficheiros tenham algum tipo de informação que possa ser convertida para uma representação textual é possível escrever uma classe que indexe essa informação [25].

4.5 *Pesquisa na rede*

Para a pesquisa na rede existem dois cenários distintos a considerar, que têm origem no facto de *peer* que inicia a pesquisa estar ou não registado com um *super peer*. O cenário mais simples coloca-se quando o *peer* não se encontra associado a um *super peer*. Neste cenário o *peer* pesquisa os seus *advertisements* a procura de outros *peers* para os quais possa enviar a sua pesquisa, criando uma lista de *peers* com os quais pode contactar. Depois envia a pesquisa para todos os *peers* na lista que criou em paralelo (cada pesquisa para um *peer* individual foi modelada como uma *thread* para ser possível este efeito) e espera os resultados, agregando os resultados da pesquisa. Este cenário é representado pela Figura 32. É importante referir novamente que os protocolos do JXTA que se encontram nos níveis inferiores criam a ilusão de que os *peers* se encontram sempre directamente ligados, mesmo quando as mensagens passam por vários *peers* intermédios.
Quando o *peer* se encontra registado num *super peer* a pesquisa é feita enviando um pedido ao *super peer*, que irá procurar em todos os indicies que possua, e que reencaminhará a pesquisa para todos os *peers* cujos índices não possua mas esteja a receber. O *super peer* é responsável por agregar os resultados recebidos com os que já possui e por os enviar de novo ao peer que iniciou a pesquisa. Este cenário é representado pela Figura 33. No caso de existir uma falha na comunicação com o *super peer* o *peer* passa automaticamente ao cenário anterior e tenta contactar todos os *peers* que conhece.

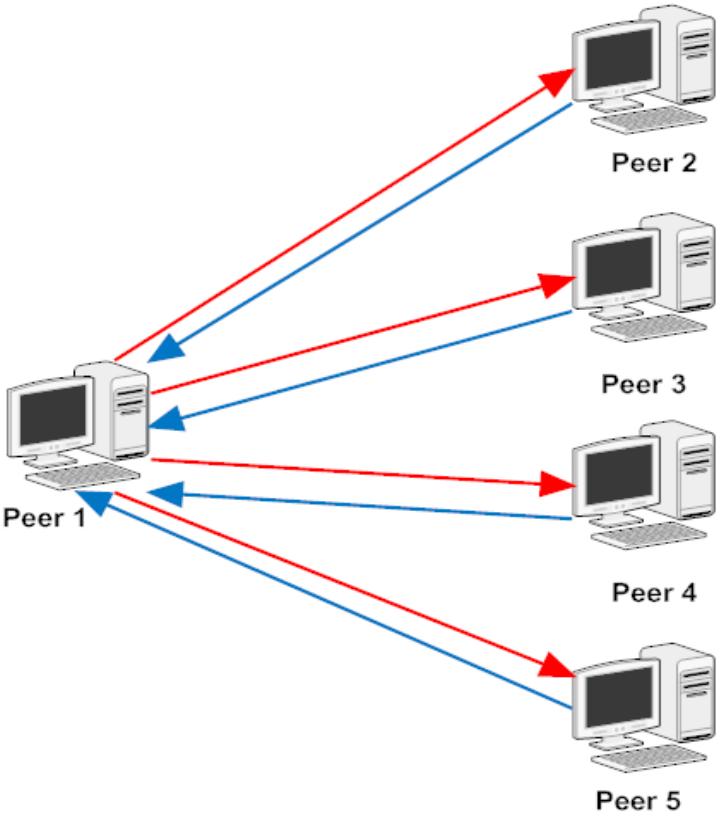

Figura 32: Pesquisa descentralizada

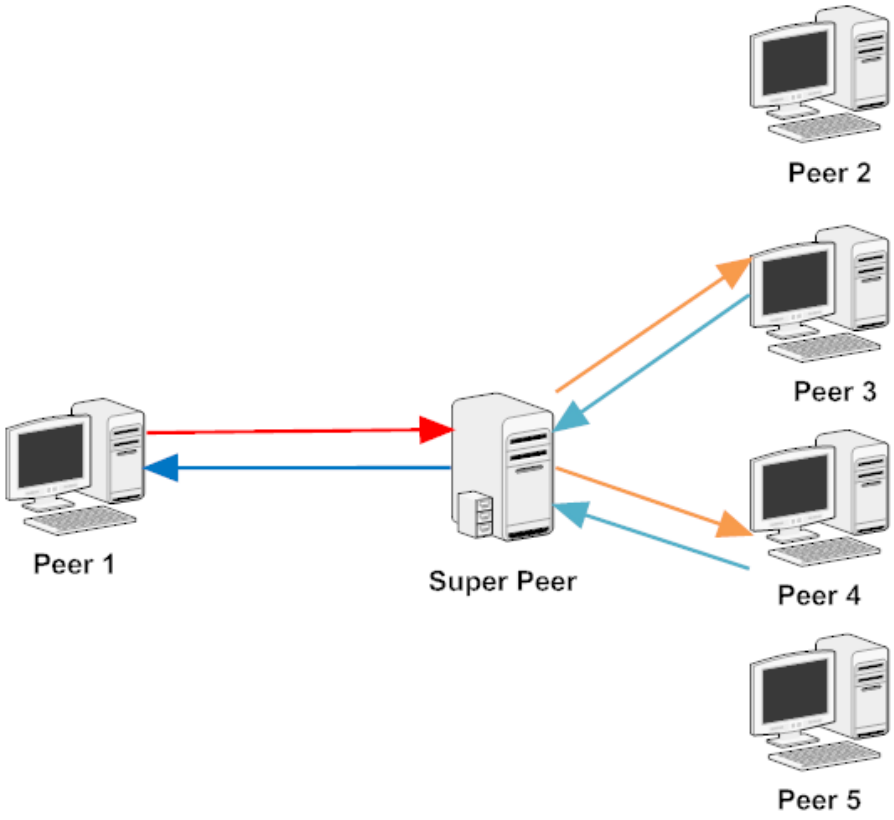

Figura 33: Pesquisa com auxílio de um super peer

Para iniciar uma pesquisa um peer deve enviar uma mensagem com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “Query”
- `<UUID>` – Este elemento deve conter o valor do identificador único atribuído ao *peer*. É utilizado nas pesquisas com recurso ao *super peer*, para que o índice do *peer* que iniciou a pesquisa não seja consultado, evitando assim devolver resultados que poderiam facilmente ser obtidos por uma pesquisa local.
- `<Query String>` – Este elemento representa a *query string* utilizada pelo Apache Lucene para efectuar as pesquisas sobre os índices, e deve seguir o formato definido pelo Lucene [26] caso contrário não vão ser produzidos resultados.

Os *peers* obtém os resultados das pesquisas através de uma sequência de mensagens em que cada mensagem contém no máximo quarenta resultados individuais. O número máximo de resultados enviados por mensagem for definido arbitrariamente tentando atingir um equilíbrio entre enviar um número razoável de resultados e não criar uma mensagem com um tamanho excessivo. Uma mensagem com quarenta resultados apresenta em média o tamanho de 9kb, o que é ainda longe do limite de 64kb imposto pelos *pipes* do JXTA.
De cada vez que recebe uma mensagem de resultados o *peer* que a recebeu deve notificar o *peer* que a enviou da correcta (ou incorrecta) recepção.
Para a devolução dos resultados a mensagem deve possuir os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “Query Response”
- `<HitCount>` – Este elemento armazena o número total de resultados obtidos pela pesquisa.

- <Result> – Este elemento agrega os campos que são devolvidos por cada resultado. Podem existir até quarenta destes elementos por mensagem, cada um contendo os seguintes elementos:
  - <FileName> – Este elemento destina-se a devolver o nome de um ficheiro encontrado na pesquisa.
  - <FileHash> – Este elemento destina-se a devolver o *hash* do ficheiro encontrado na pesquisa.
  - <UUID> – Este elemento destina-se a conter o identificador único do *peer* cujo índice contêm o resultado encontrado.

  Estes elementos correspondem directamente aos campos homónimos existentes nos índices.

- <Last Hit> – Este elemento destina-se a servir de controlo. Indica o número de sequência do último resultado enviado na mensagem. Quando o valor deste elemento for igual ao valor do elemento <HitCount> a transferência dos resultados da pesquisa estará terminada.

A mensagem de confirmação de recepção de resultados deve conter os seguintes elementos:

- <MessageType> – Este elemento define o tipo da mensagem e neste caso deve ter o valor “QueryAck”
- <Status> – O valor deste elemento explicita o estado da recepção dos resultados. Pode tomar três valores : “OK” quando os resultados foram recebidos sem erros, "Finished" quando os resultados foram recebidos sem erros e na mensagem a ser confirmada o valor do elemento <HitCount> é igual ao valor do elemento <Last Hit> e “Failed” se existiram erros na recepção dos resultados.

Uma possível sequência de mensagens resultante de uma pesquisa pode ser vista na Figura 34.

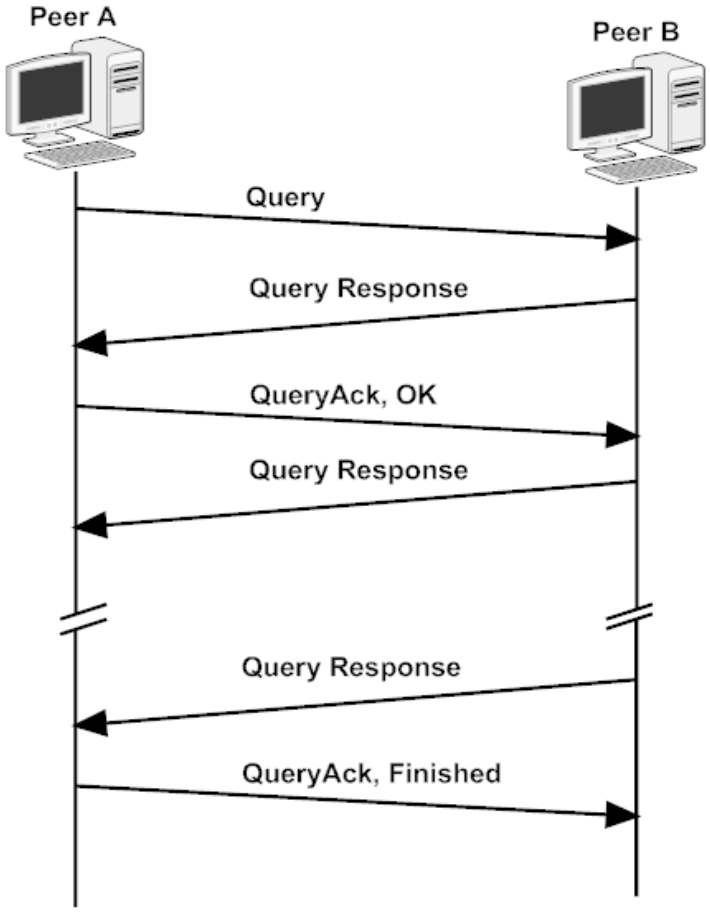

Figura 34: Sequência de mensagens resultante de uma pesquisa

4.6 *Transferência de ficheiros*

A capacidade de pesquisar por conteúdos na rede é importante, mas seria praticamente irrelevante sem uma forma de obter os conteúdos mencionados nos resultados das pesquisas. De forma a suprir esta necessidade foram criadas algumas mensagens para permitir a transferência de ficheiros entre *peers*. Nesta primeira abordagem uma transferência de um dado ficheiro é realizada entre dois *peers*, sem qualquer tentativa de aproveitamento da possível existência de outras cópias noutros *peers* mas seria desejável que evoluísse para suportar transferências com múltiplos *peers*. O processo de transferência pode começar regra geral após uma pesquisa na rede, mas nada impede que sejam feitos pedidos *ad-hoc* sem qualquer pesquisa prévia. A pesquisa prévia fornece alguma

confiança no sentido de que o ficheiro provavelmente vai estar disponível, não existindo no entanto nenhuma garantia absoluta. A mensagem que despoleta o inicío de uma transferência tem os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “FileRequest”
- `<FileHash>` – O valor deste elemento é utilizado para pesquisar o índice local do *peer* pelo ficheiro pretendido. É por isso apresentado sob a forma de uma *query* do Apache Lucene sobre o campo `FileHash` o que significa que terá a aparência de "`FileHash:<FileHash a pesquisar>`”.

A mensagem com os elementos anteriores despoleta uma pesquisa, de forma a obter a localização física do ficheiro, que se encontra indexada no campo `FilePath`. Se o ficheiro não for encontrado é enviada uma mensagem de erro, caso contrario é enviado uma mensagem de confirmação com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “TransferAccept”
- `<Total Blocks>` – O valor deste elemento define o número total de blocos que o ficheiro contém. São novamente utilizados blocos de 4kb como nas transferências de índices.

A mensagem de confirmação ao explicitar o número de blocos de um ficheiro sem no entanto enviar ainda nenhum bloco permite que o *peer* que requisitou a transferência assuma o controlo e que requisite os blocos que quiser pela ordem que quiser, o que abre as portas à eventual implementação de transferências de com múltiplos *peers*.

Para pedir um bloco o *peer* utiliza uma mensagem com o seguinte formato:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “BlockRequest”
- `<Block>` – O valor deste elemento define o bloco que o *peer* quer receber.

A resposta deverá ser uma mensagem com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “DataChunk”
- `<ByteCount>` – Este elemento diz quantos bytes são efectivamente enviados no elemento `<Datablock>`.
- `<Datablock>` – Este elemento contem os dados sob a forma de um *array* de *bytes*, com tamanho especificado pelo elemento `<ByteCount>`

Quando for recebido com sucesso o último bloco, ou quando existir um erro durante a transferência o *peer* deve enviar uma mensagem com os seguintes elementos:

- `<MessageType>` – Este elemento define o tipo da mensagem e neste caso deve ter o valor “Transfer Info”
  `<Status>` – Este elemento informa o *peer* que recebe a mensagem de sobre algo que aconteceu durante a transferência. Neste momento estão definidos os valores “Finished” para quando a transferência deve terminar (com sucesso) e “Abort” para quando a transferência deve ser abortada.

Uma possível sequência de mensagens resultante da transferência de um ficheiro pode ser vista na Figura 35.

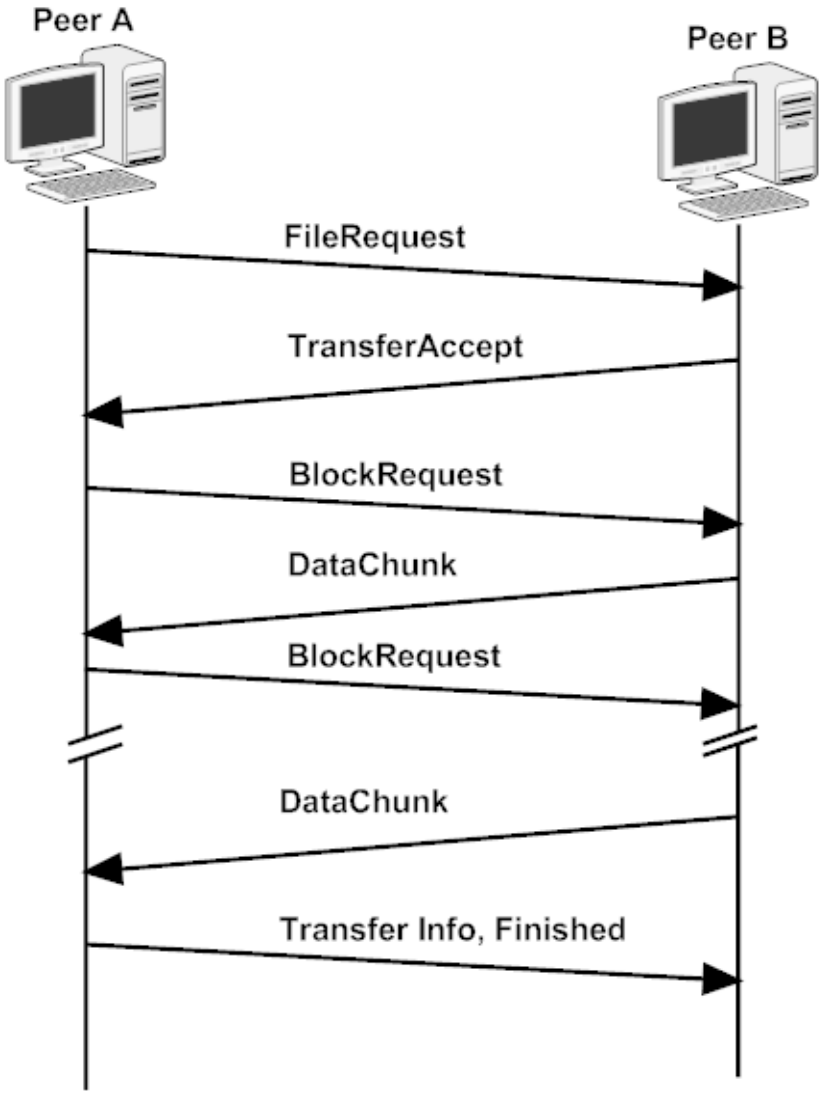

Figura 35: Sequência de mensagens resultante da transferência de um ficheiro

4.7 *Disponibilização de Web Services na rede*

Uma rede *peer-to-peer* que oferece pesquisa e transferência de ficheiros pode ser considerada algo banal dada a popularização deste tipo de redes. No entanto, uma rede *peer-to-peer* cujos *peers,* para além das transferências de ficheiros também podem alojar serviços, não é tão comum. Os *peers* não devem estar limitados a um conjunto de serviços pré-existentes e deve ser fácil acrescentar serviços. Para tal foi desenhado um mecanismo do tipo *plugins* baseado num ficheiro de propriedades, de forma a ser possível com o mínimo de esforço desenvolver serviços e depois disponibiliza-los num *peer* que já se encontra

ligado a rede. Desta forma acrescentar um novo serviço à lista de serviços que um *peer* pode oferecer é tão simples como acrescentar uma entrada ao ficheiro "services.properties" e garantir que a classe nessa entrada se encontra no *classpath* do Java.

Foi definido que os serviços disponibilizados desta forma seriam do tipo *WebService*, o que implica neste caso que vão respeitar as especificações definidas pela biblioteca JXTA-SOAP. Como tal para cada serviço deve existir um WSDL cuja geração é recomendado que seja feito pelo Apache Axis (apesar de nada impedir a utilização de outra forma para gerar o WSDL). De forma a uniformizar as formas de iniciar o serviço e de o configurar foi definida uma interface que os serviços têm de implementar:

```
public interface JxtaServicesInterface
{
    public String getWsdlPath();
    public ServiceDescriptor getDescriptor();
    public void Configure();
}
```

Figura 36: Interface definida para um serviço

- `getWsdlPath()` – Este método devolve a localização do WSDL que representa o serviço. Foi definida para que quem desenvolva um serviço possa colocar o WSDL onde achar mais conveniente.
- `GetDescriptor()` – Este método existe para oferecer um acesso fácil ao *service descriptor*, um componente chave na criação de serviços com o JXTA-SOAP. Este *service descriptor* deve ser preenchido correctamente por quem desenvolva um serviço. Um exemplo comentado pode ser visto na Figura 37,
- `Configure()` – Este método existe para permitir a configuração do serviço. É dada total liberdade sobre a forma como o serviço deve ser configurado para que quem desenvolva um serviço escolha o método que ache mais adequado.

```
private static final ServiceDescriptor DESCRIPTOR =
       new ServiceDescriptor(
       "pt.ua.JxtaIndexPeer.JxtaServices.Tiff2Jpg",// class
       "Tiff2JpgService",                          // service name
       "0.1",                                      // version
       "Marco Pereira",                            // creator
       "jxta:/a.very.unique.spec/uri",             // specURI
       "Store and index a file",                   // description
       "urn:jxta:jxta-NetGroup",                   // peergroup ID
       "JXTA NetPeerGroup",                        // PeerGroup name
       "JXTA NetPeerGroup",                        // PeerGroup
                                                   // description
       false,                                      // secure policy flag
                                                   //(use default=false)
       null);                                      // security policy
                                                   // type (use no
                                                   // policy)
```

Figura 37: Exemplo de um *service descriptor* do JXTA-SOAP

É recomendado que quando forem utilizadas as ferramentas do Apache Axis para gerar o WSDL de um serviço estas sejam utilizadas antes da implementação da interface da Figura 36 para evitar expor no WSDL gerado os métodos da referida interface.

Os serviços disponibilizados desta forma são representados na rede por *module advertisements* que os clientes devem procurar pelos métodos normais do JXTA.

Os clientes para os serviços desenvolvidos adoptam também a mesma estratégia do tipo *plugin* utilizando porem o ficheiro “client.properties”. Os clientes também possuem uma interface definida que devem respeitar. Esta interface serve apenas para garantir um método comum de inicialização, sendo que todo o controlo sobre a utilização de funcionalidades da rede deixa de ser da responsabilidade do *peer* para passar a ser da responsabilidade do cliente do serviço.

```
public interface JxtaServicesClientInterface
{
    public void start();
}
```

Figura 38: Interface definida para um serviço

A utilização do JXTA-SOAP cria no entanto algumas limitações. Neste momento para criar um serviço que tenha como parâmetro de entrada um ficheiro, todo o conteúdo do ficheiro tem de ser transferido como um *array* de bytes dentro de uma mensagem SOAP, o que é uma forma extremamente ineficaz de transferir

um ficheiro. Pior ainda, a implementação utilizada não consegue serializar desta forma ficheiros com tamanho superior a 40mb (a máquina virtual do Java fica sem memoria disponível e o processo é abortado) o que pode conduzir a que alguns serviços nunca recebam alguns pedidos a eles dirigidos. A solução a adoptar para contornar este problema seria utilizar o mecanismo de *attachments* disponibilizado pelo Apache Axis, no entanto todas as tentativas de o utilizar em conjunto com a biblioteca JXTA-SOAP deram origem a excepções quando se tentava adicionar um *attachment* ao pedido enviado para o servidor.

4.8 *Problemas da plataforma*

A versão da implementação em Java do JXTA escolhida para ser utilizada neste trabalho foi a 2.5, cujo lançamento se deu em Novembro de 2007. Com esta versão ocorreu uma mudança no funcionamento interno, donde as operações de rede deixaram de ser baseadas nas operações da biblioteca `java.net.Socket` para passar a ser baseadas nas operações da biblioteca Native Input/Output (NIO)[27] `java.nio.SocketChannel`. Esta mudança introduziu alguns erros estranhos que estão actualmente ainda a ser corrigidos nas *nightly builds* do JXTA. Como exemplo temos um erro que impedia os pipes do JXTA de serem correctamente fechados que só foi corrigido na *nightly build* de 6 de Abril de 2008 [28]. Relacionado com a utilização do NIO foi detectado um problema para qual não foi encontrada uma solução. Quando um *peer* decide sair da rede (por exemplo quando a aplicação é fechada) os restantes *peers* lançam imediatamente uma excepção dado que perderam a comunicação com esse *peer*, comunicação essa que é criada automaticamente e gerida pelo JXTA quando um *peer* se junta a rede. As consequências detectadas desta excepção variam entre a simples mensagem de aviso escrita para a consola até a inutilização do *peer*.

Um problema que aparecia de forma frequente era a perda de mensagens. Um *peer* enviava uma mensagem para outro e esta não era recebida. A mensagem que era perdida era sempre a primeira mensagem a ser enviada após a abertura de um canal de comunicação. Ao ser sempre a primeira mensagem a ser perdida suspeitou-se que poderia existir um problema com a fila em que as mensagens são armazenadas até ser definido um *message listener* para o pipe (a fila de

espera é uma estrutura interna do BidiPipe). Para contornar este problema foi introduzido um atraso de 500 millisegundos no envio da primeira mensagem, atraso esse que permite a atribuição de um *message handler* ao pipe evitando assim a fila de espera. Este pequeno truque não pode ser considerado uma solução permanente mas ajuda a minorar o problema. No futuro devem ser efectuadas diligências no sentido de determinar se o problema se encontra realmente na fila de espera ou se é causado pela incorrecta utilização dos BidiPipes neste trabalho.

5 Testes

Foram desenhados alguns testes de pequena escala para ajudar analisar o desempenho inicial e o comportamento da rede. Para a realização dos testes foi criada uma pequena rede apenas com três *peers*. Se os resultados forem satisfatórios deve ser equacionada a realização de testes em larga escala com um número muito superior de *peers*.

5.1 *Conversão Tiff para Jpeg*

O primeiro teste foi desenhado para testar o tempo que demora utilizar um serviço da rede para converter imagens que se encontram formato Tiff para o formato Jpeg. O serviço de conversão encaixa-se no perfil dos serviços que podem ser criados para suportar bibliotecas digitais, criando assim um teste realista. Este teste visa tentar perceber se com *peers* suficientes na rede e com uma estratégia de divisão de tarefas a conversão de uma grande quantidade de imagens efectuada por um serviço residente na rede consegue competir com a conversão da mesma quantidade de imagens efectuada num só *peer*.

Para este teste foram utilizadas imagens geradas a partir de um pdf. A aplicação utilizada para gerar as imagens foi o ImageMagick[29]. Cada imagem gerada tem o tamanho de 980kb. Foram realizados testes de conversão com 10, 20 e 30 imagens, tendo cada teste sido repetido para reflectir a adição de um *peer* que corre o serviço a rede. Cada teste é composto por dez ensaios distintos de forma a ser possível estabelecer uma média dos tempos dos ensaios. Para cada teste existe um tempo de referência obtido ao cronometrar uma chamada directa ao código que efectua a conversão sem utilizar o serviço disponível na rede. Este tempo de referência representa o tempo de conversão das imagens numa única máquina isolada.

O serviço de conversão pode converter as imagens que recebe de duas formas distintas: pode tentar utilizar a implementação interna baseada na biblioteca JAI (Java Advanced Imaging) [30] ou pode tentar utilizar uma aplicação externa. Visto que a implementação interna não é ainda capaz de lidar com o formato do Tiff

gerado pelo ImageMagick (neste momento a biblioteca JAI não suporta a conversão para Jpeg imagens com mais de três canais de informação) foi decidido utilizar uma aplicação externa cuja escolha recaiu novamente no ImageMagick.

5.1.1 Conversão de 10 imagens

#Peers	Tempo (em millisegundos)									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
1	11265	7359	6500	6703	6984	6656	6797	6593	7312	7078
2	9390	4907	4827	4899	4577	4656	4173	4473	4479	4574
3*	8413	4458	3995	4556	4365	3927	3999	3921	4417	5671

Tabela 2: Tempos de conversão de 10 imagens

#Peers	Tempo (em millisegundos)
1	7324,7
2	5095,5
3*	4772,2

Tabela 3: Média dos tempos de conversão de 10 imagens

* O *peer* que está a efectuar os pedidos também disponibiliza o serviço

Tempo de referência para conversão de 10 imagens: 3437 ms.

5.1.2 Conversão de 20 imagens

#Peers	Tempo (em millisegundos)									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
1	20063	15750	13422	13109	13453	14937	17672	20422	18219	13563
2	11228	8383	8504	8869	8720	8580	8831	8862	8824	8502
3*	8737	8220	7710	8456	10477	7519	7566	7691	8210	7883

Tabela 4: Tempos de conversão de 20 imagens

#*Peers*	Tempo (em millisegundos)
1	16061
2	8930,3
3*	8246,9

Tabela 5: Média dos tempos de conversão de 20 imagens

* O *peer* que está a efectuar os pedidos também disponibiliza o serviço

Tempo de referência para conversão de 20 imagens: 7344 ms.

5.1.3 Conversão de 30 imagens

#*Peers*	Tempo (em millisegundos)									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
1	22860	19922	25375	18953	19250	22719	18594	19016	23312	19875
2	16708	12413	12683	12803	12386	12780	11839	12295	12922	13485
3*	14354	13022	11545	11947	11778	11294	12386	11934	12433	13188

Tabela 6: Tempos de conversão de 30 imagens

#*Peers*	Tempo (em millisegundos)
1	20987,6
2	13031,4
3*	12388,1

Tabela 7: Média dos tempos de conversão de 30 imagens

* O *peer* que está a efectuar os pedidos também disponibiliza o serviço

Tempo de referência para conversão de 30 imagens: 11094 ms.

5.1.4 Conclusões do teste

Pelos resultados pode verificar-se claramente que à medida que o número de *peers* que disponibilizam o serviço aumenta o tempo necessário para converter as imagens diminui aproximando-se bastante do tempo de referência. É expectável que com mais *peers* e para mais imagens seja vantajoso utilizar este tipo de serviço em vez de efectuar as conversões todas na máquina local. Nota-se também que o tempo de conversão mais elevado corresponde normalmente ao

primeiro ensaio. Isto indica que existe um custo associado a chamar o serviço pela primeira vez (provavelmente derivado de inicializações que tem de ser feitas pelo Axis).

5.2 *Pesquisa na rede*

Este teste foi desenhado para testar o desempenho da rede a efectuar pesquisas. Dado o número limitado de peers disponíveis não se espera existir uma grande diferença no desempenho entre pesquisas que sejam dirigidas a um *super peer* e pesquisas que sejam difundidas pela rede, por isso apenas estas últimas são objecto deste teste. Para este teste os peers devem indexar um número pré determinado de ficheiros. São efectuadas pesquisas de forma a obter resultados e é recolhido o tempo que demora cada pesquisa. Cada teste é composto por 10 ensaios independentes de forma a ser possível estabelecer uma média do tempo de pesquisa.

5.2.1 Pesquisa que não devolva resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
969	734	594	672	625	593	641	593	641	593	665,5

Tabela 8: Tempos de pesquisa para não obter resultados

5.2.2 Pesquisa que devolve 10 resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
687	954	625	609	625	625	594	579	656	578	653,2

Tabela 9: Tempos de pesquisa para obter 10 resultados

5.2.3 Pesquisa que devolve 40 resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
704	625	656	609	907	672	625	594	640	610	664,2

Tabela 10: Tempos de pesquisa para obter 40 resultados

5.2.4 Pesquisa que devolve 100 resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
672	688	656	703	687	735	844	657	641	734	701,7

Tabela 11: Tempos de pesquisa para obter 100 resultados

5.2.5 Pesquisa que devolve 300 resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
1110	922	905	828	859	735	907	781	734	766	854,7

Tabela 12: Tempos de pesquisa para obter 300 resultados

5.2.6 Pesquisa que devolve 8378 resultados

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
7328	6540	6500	6391	5843	6547	5797	6047	6797	6797	6458,7

Tabela 13: Tempos de pesquisa para obter 8378 resultados

5.2.7 Conclusões do teste

Pelos resultados pode concluir-se que pesquisar a rede para obter um pequeno número de resultados é uma operação rápida. O teste em que não se espera obter resultados e os testes onde se espera obter 10 e 40 resultados resultam todos apenas na criação de uma única mensagem de resposta e por isso apresentam tempos de pesquisa semelhantes. É importante notar que em média levou mais tempo a efectuar uma pesquisa que não devolve resultados do que a efectuar uma que devolva 10 ou 40 resultados. Esta aparente contradição deve-se a flutuações no tráfego que passava pela rede na altura em que foram efectuados os testes. Ao aumentar o número de resultados esperados também aumenta o tempo que leva a efectuar uma pesquisa. O último teste mostra um tempo médio de 6,5 segundos para efectuar a pesquisa. Este tempo refere-se à obtenção da totalidade dos resultados, que são depois apresentados ao utilizador. Nenhum sistema em produção deve demorar tanto tempo a apresentar os resultados de uma pesquisa. Visto que os resultados de uma pesquisa são recebidos em mensagens com quarenta resultados cada é possível mascarar este problema se em vez de se esperar pelo conjunto de resultados completo se começar a mostrar ao utilizador os resultados da pesquisa logo que a primeira mensagem com resultados é recebida, actualizando os resultados que são mostrados ao utilizador à medida que são recebidas mais mensagens.

5.3 *Transferência de índices*

Este teste foi desenhado para descobrir quanto tempo leva um índice a migrar de um *peer* para um *super peer*. A transferência de um índice é equivalente à transferência de um ficheiro de tamanho igual (a diferença está apenas em algumas questões de controlo). Foram efectuados três testes com índices de diferentes tamanhos, enviados de dois *peers* para um *super peer*. Cada teste é composto por 10 ensaios distintos de forma a ser possível estabelecer uma média do tempo de transferência.

5.3.1 Índice Vazio

O índice utilizado neste teste foi um índice em branco gerado automaticamente quando o *peer* é executado pela primeira vez. Tem um tamanho de 40 bytes.

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
718	640	656	688	641	672	656	672	625	641	660,9

Tabela 14: Tempos de transferência de um índice vazio

5.3.2 Índice com 1.75 Mbytes

O índice utilizado neste teste foi gerado a partir de 294 ficheiros de texto.

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
4157	3031	2797	2766	2734	2984	2813	2718	2641	2625	2926,6

Tabela 15: Tempos de transferência de um índice com 1.75Mbytes

5.3.3 Índice com 12.7Mbytes

O índice utilizado neste teste foi gerado a partir de 8278 ficheiros de texto.

Tempo (em millisegundos)										
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Média
15531	12641	15109	18765	13359	13422	12719	13235	12766	18172	14571,9

Tabela 16: Tempos de transferência de um índice com 12.7Mbytes

5.3.4 Conclusões do teste

Com este teste é possível notar-se que como seria de esperar à medida que o tamanho do índice aumenta também aumenta o seu tempo de transferência. Os tempos de transferência podem sofrer variações superiores se a carga na rede aumentar de forma drástica.

Dado que o tempo de transferência de um índice com pouco mais de uma dezena de *megabytes* de tamanho se encontra já nas dezenas de segundos é expectável que para índices de tamanho muito superior cujos tempos de transferência sejam muito longos que caso apareça uma pesquisa dirigida ao *super peer* este faça uso dos mecanismos que lhe permitem propagar a pesquisa para os *peers* que ainda estão a transferir os seus índices de forma a obter um conjunto de resultados completo. É importante reflectir até que ponto a migração de índices com tamanhos consideráveis compensa. Um índice com várias centenas de *megabytes* irá necessariamente provocar uma transferência demorada e vai consumir recursos do *super peer* (largura de banda e espaço de armazenamento). De forma a manter todas as opções em aberto o protocolo de registo num *super peer* deixa ao cuidado de cada *peer* se quer ou não transferir o seu índice para o *super peer*. Assim poderá ser criado do lado dos *peers* clientes um limite para o tamanho máximo que um índice pode ter para ser elegível para transferência para o *super peer*. De forma a não serem perdidos resultados as pesquisas devem ser propagadas para todos os *peers* que o *super peer* conheça e para os quais não tenha índice, em vez de serem só propagadas para os *peers* que estão ainda a transferir o seu índice.

6 Conclusões

A utilização de redes *peer-to-peer* para ajudar ao funcionamento de bibliotecas digitais é viável. A maioria dos sistemas *peer-to-peer,* cujo objectivo é apenas a transferência de ficheiros entre os seus utilizadores, consome vastas quantidades de espaço de armazenamento desperdiçando ciclos de processamento. Por outro lado existem projectos de computação distribuída que consomem ciclos de processamento mas praticamente não usam o espaço de armazenamento que possuem. Com a arquitectura proposta e com uma cuidada distribuição dos serviços pelos *peers* que compõem a rede seria possível rentabilizar os recursos que não estão a ser utilizados nos cenários anteriores e esta é uma perspectiva aliciante.

A arquitectura proposta neste trabalho apresenta ainda muitas falhas e problemas que tem de ser resolvidos antes de se pensar numa utilização real. Deve ser expandida de forma a poder ser utilizada fora da rede local onde foi testada e cada problema identificado deve ser solucionado. Pode dizer-se que é um projecto ainda na sua infância mas que apresenta um grande potencial de crescimento.

6.1 *Trabalho Futuro*

Para o potencial deste trabalho ser completamente aproveitado são necessários ainda melhoramentos e refinamentos.

A prioridade principal deverá ser chegar a uma solução definitiva para o problema da perda de mensagens descrito na secção 4.9. Depois devem ser consideradas quaisquer opções que não limitem o tamanho dos ficheiros que podem ser transferidos quando são utilizados *webservices* criados com a combinação Axis/JXTA-SOAP.

Deve ser considerada a utilização da rede pública de rendezvous JXTA para ser possível interligar vários *peers* que não se encontrem na mesma rede local. Alcançar este objectivo vai requerer um estudo sobre a utilização e criação de um

super peer do tipo *relay* para ajudar a contornar obstáculos existentes na rede, tais como *firewalls* ou NATs. Este estudo será válido também no caso de se criar uma rede privada de *rendezvous*.

Seria também interessante criar um peer group privado exclusivo para os peers dedicados a fornecer pesquisas e serviços de apoio a bibliotecas digitais. A utilização de um peer group privado pode abrir caminho á criação de um ambiente que possa garantir algum nível de segurança para aplicações críticas.

Referências

- 1: Andy Oram, "Peer-to-Peer : Harnessing the Power of Disruptive Technologies", O'Reilly, ISBN 059600110X, 2001
- 2: Alfred Wai-Sing Loo, "Peer-to-Peer Computing: Building Supercomputers with Web Technologies", Springer, ISBN 1846283817, 2006
- 3: Ian J. Taylor et al, "From P2P to Web Services and Grids: Peers in a Client/Server World", Springer, ISBN 1852338695, 2004
- 4: Clay Shirky, "What is P2P... And What Isn't?", 2000, http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html
- 5: Nelson Minar, "Distributed Systems Topologies: Part 1", 2001, http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html
- 6: Nelson Minar, "Distributed Systems Topologies: Part 2", 2001, http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html
- 7: John Rissom and Tim Moors, "Survey of Research towards Robust Peer-to-Peer Networks: Search Methods", 2006,http://www.ee.unsw.edu.au/~timm/pubs/06compnet/submitted.pdf
- 8: Yoram Kulbak and Danny Bickson, "The eMule Protocol Specification", 2005, http://www.cs.huji.ac.il/labs/danss/presentations/emule.pdf
- 9: Brendon J. Wilson, "JXTA", New Riders Publishing, ISBN 0735712344, 2002
- 10: , "The Gnutella Protocol Specification v0.41", , http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf
- 11: Tor Klingberg and Raphael Manfredi, "Gnutella 0.6", 2002, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html
- 12: Anurag Singla and Christopher Rohrs, "Ultrapeers: Another Step Towards Gnutella Scalability", , http://www.limewire.com/developer/Ultrapeers.html
- 13: Jem E. Berkes, "Decentralized Peer-to-Peer Network Architecture:Gnutella and Freenet", 2003, www.sysdesign.ca/archive/berkes_gnutella_freenet.pdf
- 14: , "JXTA Homepage", , https://jxta.dev.java.net/
- 15: Sun Microsystems, "JXTA Java™ Standard Edition v2.5: Programmers Guide", https://jxta-guide.dev.java.net/source/browse/*checkout*/jxta-guide/trunk/src/guide_v2.5/JXSE_ProgGuide_v2.5.pdf, 2007
- 16: , "JXTA v2.0 Protocols Specification", https://jxta-spec.dev.java.net/JXTAProtocols.pdf, 2007
- 17: W3C, "Extensible Markup Language (XML) 1.0", 2006, http://www.w3.org/TR/xml/
- 18: Michele Amoretti, Francesco Zanichelli and Gianni Conte, "Enabling Peer-To-Peer Web Service Arquitectures With JXTA-SOAP", 2008, https://soap.dev.java.net/amorettiESociety08paper.pdf
- 19: , "JXTA-SOAP Project", , https://soap.dev.java.net/
- 20: W3C, "Simple Object Access Protocol (SOAP) 1.1", 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
- 21: W3C, "Web Services Description Language (WSDL) 1.1", 2001, http://www.w3.org/TR/wsdl
- 22: , "Apache Axis", , http://ws.apache.org/axis/

- 23: , "Apache Lucene", , http://lucene.apache.org/java/docs/index.html
- 24: P. Leach et al, "A Universally Unique IDentifier (UUID) URN Namespace", 2005, http://tools.ietf.org/html/rfc4122
- 25: Otis Gospodnetic, Erik Hatcher, "Lucene In Action", Manning Publications, ISBN 1932394281, 2004
- 26: , "Query Parser Syntax", 2007, http://lucene.apache.org/java/2_2_0/queryparsersyntax.html
- 27: Mike Duigou, "JXSE 2.5 : What's Cool #1 -- NIO TCP : JXTA", 2007, http://blogs.sun.com/bondolo/category/JXTA?page=2
- 28: , "Java.net JXTA Community Forum", 2008, http://forums.java.net/jive/thread.jspa?threadID=39079&tstart=135
- 29: , "ImageMagick Homepage", , http://www.imagemagick.org
- 30: , "Java Advanced Imaging API Home Page", , (http://java.sun.com/products/java-media/jai/iio.html

Anexos

Anexo A - Criação de um ficheiro WSDL com recurso ao Netbeans e ao Apache Axis

O ambiente de desenvolvimento utilizado para este trabalho foi o Netbeans 6. Este ambiente de desenvolvimento não possui uma integração nativa com o Apache Axis, sendo por isso necessárias algumas configurações para permitir a criação de um WSDL a partir do ambiente de desenvolvimento.

O primeiro passo é instalar o Apache Axis 1.4. No caso foi escolhido instalar na directoria g:\axis-1_4.
O segundo passo é editar o ficheiro build.xml do projecto JxtaIndexPeer. Este ficheiro pode ser editado directamente do ambiente de desenvolvimento. O objectivo desta edição é acrescentar um alvo para que manualmente possa ser gerado o WSDL a partir de uma classe java existente. No caso foi escolhido como base a classe com o nome de Tiff2Jpg. Antes da linha final do ficheiro build.xml deve ser acrescentado o seguinte código:

```
<!-- Web Service Name-->
    <property name="service.name" value="Tiff2JpgService"/>

    <!-- Web Service Namespace Used in the WSDL -->
    <property name="service.namespace" value="http://JxtaIndexPeer.pt"/>

    <!-- Java Package Name -->
    <property name="service.java.package"
value="pt.ua.JxtaIndexPeer.JxtaServices"/>

    <!-- Name of Java Implementation Class-->
    <property name="service.java.name" value="Tiff2Jpg"/>

    <!-- Physical Location of Java Implementation -->
    <property name="service.java.path"
value="java\pt\ua\JxtaIndexPeer\JxtaServices"/>

    <!-- The URL of the Web SERVER + Application Name -->
    <property name="service.url" value="http://localhost:8080/ $
{ant.project.name} "/>

    <!-- **** Don't edit these ***-->
    <!-- Complete Physical Path + Java Class Name-->
    <property name="service.java"
              value="${basedir}\src\${service.java.path}\$
{service.java.name}.java"/>

    <!-- Java Class Name with Package Information-->
    <property name="service.class"
        value="${service.java.package}.${service.java.name}"/>

    <!-- Location where WSDL will be Generated-->
    <property name="service.wsdl"
        value="${basedir}/${service.name}.wsdl"/>

    <!-- Location of Deployment Descriptor-->
    <property name="service.deploy.wsdd"
        value="${basedir}/ServiceDeploy.wsdd"/>

    <!-- Location of Undeployment Descriptor-->
    <property name="service.undeploy.wsdd"
        value="${basedir}/${service.name}.undeploy.wsdd"/>

    <!-- Where the compiled java classes are -->
    <property name="build.classes"
        value="${basedir}/build/classes"/>

    <!-- The Admin Service for Deployment/Undeployment -->
    <property name="axis.admin-service"
        value="${service.url}/services/AdminService"/>

    <!-- The ENDPoint of the Web Service-->
    <property name="service.default.url"
        value="${service.url}/services/${service.name}"/>
```

```
    <!-- Axis Home -->
    <property name="axis.home" value="G:\axis-1_4" />

    <path id="base.libraries">
        <fileset dir="${axis.home}/lib">
            <include name="*.jar" />
        </fileset>
    </path>

    <property name="classpath_id" value="base.libraries" />

    <target name="validate">
        <available property="classpath_id" value="base.libraries" file="$
{axis.home}axis-ant.jar"/>
    </target>

    <taskdef name="java2wsdl"
classname="org.apache.axis.tools.ant.wsdl.Java2WsdlAntTask"
             classpathref="${classpath_id}"/>

    <taskdef name="wsdl2java"
classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask"
             classpathref="${classpath_id}"/>

    <target name="MY_Java2WSDL">
        <java2wsdl
                    classname="${service.class}"
                    location="${service.default.url}"
                    namespace="${service.namespace}"
                    output="${service.wsdl}"
                    porttypename="${service.name}PortType"
                    bindingname="${service.name}Binding"
                    serviceelementname="${service.name}Service"
                    serviceportname="${service.name}Port"
                    stopclasses="java.*,javax.*,org.apache.axis.AxisFault"
                    style="document" typemappingversion="1.1"
use="literal"
                    useinheritedmethods="false" soapaction="DEFAULT">
            <classpath>
                <pathelement location="${build.classes}" />
            </classpath>
        </java2wsdl>
    </target>
```

Este código deve ser adaptado as necessidades de cada situação específica. Para tal só devem ser alterados os valores das seis primeiras propriedades definidas e da localização do Apache Axis.

Para gerar o WSDL o projecto necessita de ser compilado. Depois é necessário correr o alvo definido (neste caso com o nome de “MY_Java2WSDL”) para finalmente geral o WSDL. Assumindo que as alterações ao ficheiro build.xml foram efectuadas no ambiente de desenvolvimento este alvo deve aparecer num

painel a esquerda, bastando depois carregar nele com o botão direito do rato e escolher a opção "*Run Target*" (Figura 39)

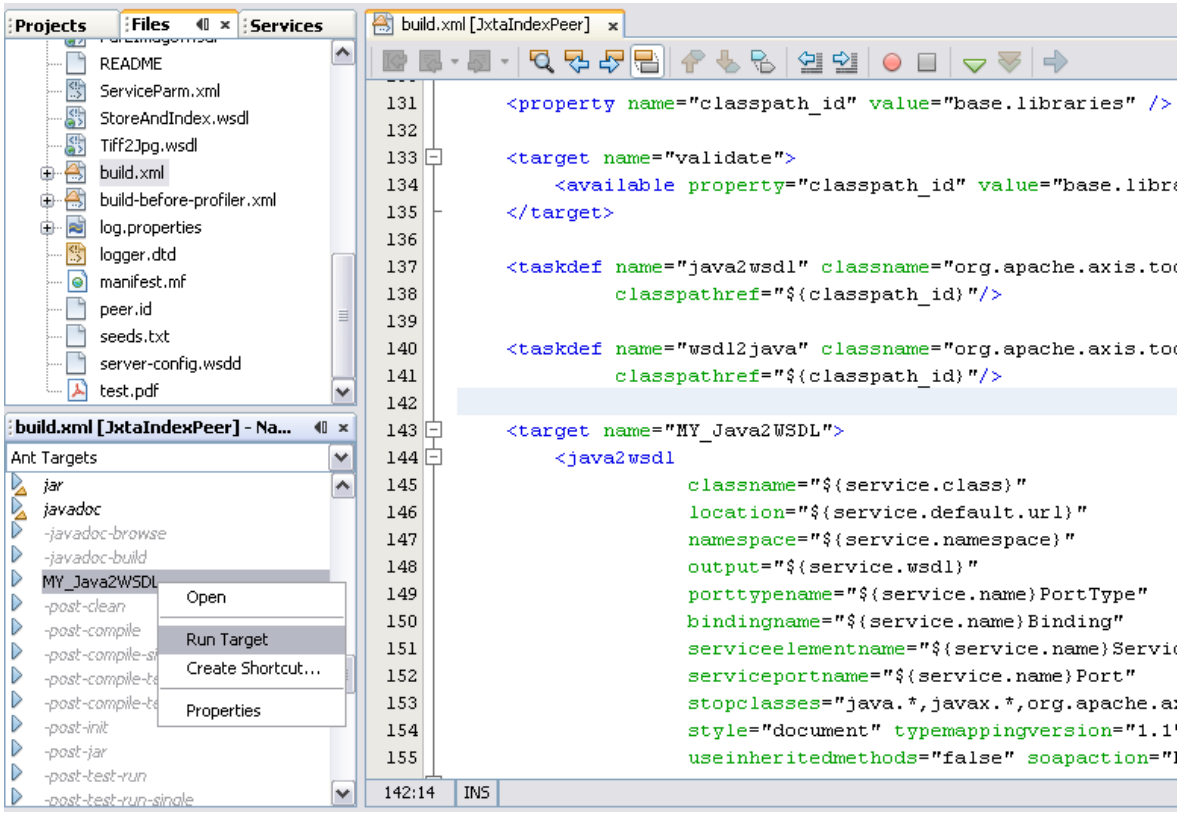

Figura 39: Passo final da criação de um WSDL

O WSDL é criado na directoria raíz do projecto. Este deve depois ser copiado para junto do arquivo .jar que é utilizado para correr o projecto (que se encontra na directoria dist).

Anexo B - Bibliotecas necessárias para o projecto

Este anexo lista as bibliotecas necessárias para a correcta compilação e execução do projecto. Estas bibliotecas devem ser geridas pelo Netbeans. Para criar uma referência a uma biblioteca no Netbeans 6 devem seguir-se os seguintes passos:

- Seleccionar a opção “Libraries” disponível no menu “Tools”.
- Escolher a opção “New Library...” e dar um nome apropriado a biblioteca no dialogo que irá a aparecer.
- Adicionar os ficheiros necessários à biblioteca criada (recomenda-se que se adicionem ficheiros .jar individuais e não pastas inteiras).

Lista de bibliotecas necessárias:

- JAI
  - clibwrapper_jiio.jar
  - jai_codec.jar
  - jai_core.jar
  - jai_imageio.jar
  - jai.mlibwrapper_jai.jar
- JID3
  - JID3.jar
- JXTA
  - bcprov-jdk14.jar
  - javax.servlet.jar
  - jxta.jar
  - jxtasecurity.jar
  - jxtashell.jar
  - org.mortbay.jetty.jar
- JXTA-SOAP
  - activation.jar
  - addressing-1.0.jar

    - axis.jar
    - axis-ant.jar
    - commons-discovery-0.2.jar
    - commons-logging-1.0.4.jar
    - jaxrpc.jar
    - jxta-soap.jar
    - log4j-1.2.8.jar
    - mail.jar
    - opensaml-1.0.1.jar
    - saaj.jar
    - wsdl4j-1.5.1.jar
    - wss4j.jar
    - xalan-2.6.0.jar
    - xercesImpl.jar
    - xml-apis.jar
    - xmlsec-1.2.1.jar
- Lucene
    - lucene-core-2.2.0.jar
    - lucene-queries-2.2.0.jar
- PDFBox
    - FontBox-0.1.0.jar
    - JempBox-0.2.0.jar
    - PDFBox-0.7.3.jar

Anexo C – Configurar um peer para se comportar como super peer

Para configurar um *peer* para se comportar como um *super peer* basta passar o argumento de linha de comandos “server”. O comando para executar a aplicação ficaria então:

```
java -jar JxtaIndexPeer.jar server
```

Se não for especificado nenhum argumento o *peer* comporta-se como um *peer* normal.

A interface gráfica disponibilizada por cada tipo de peer é diferente. A interface de um peer normal (Figura 40) permite efectuar pesquisas na rede, indexar conteúdo e instanciar e usar serviços. A interface de um super peer (Figura 41) permite monitorizar os peers que se encontram ligados ao super peer e instanciar e usar serviços.

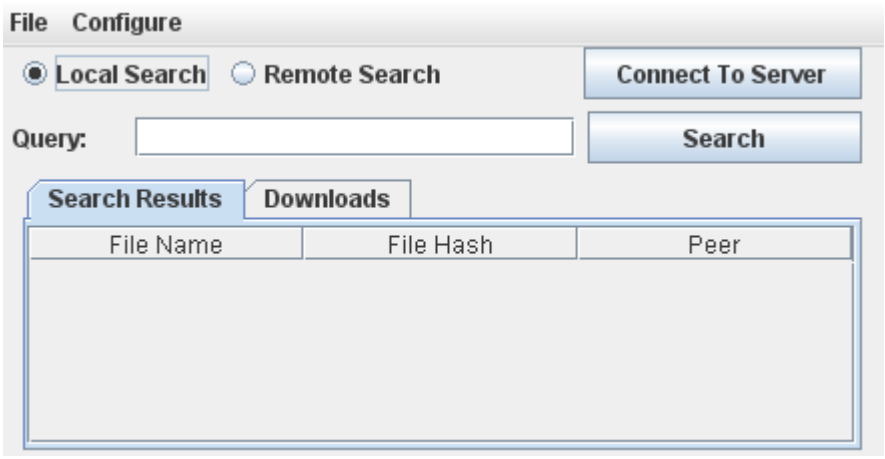

Figura 40: Interface de um *peer* normal

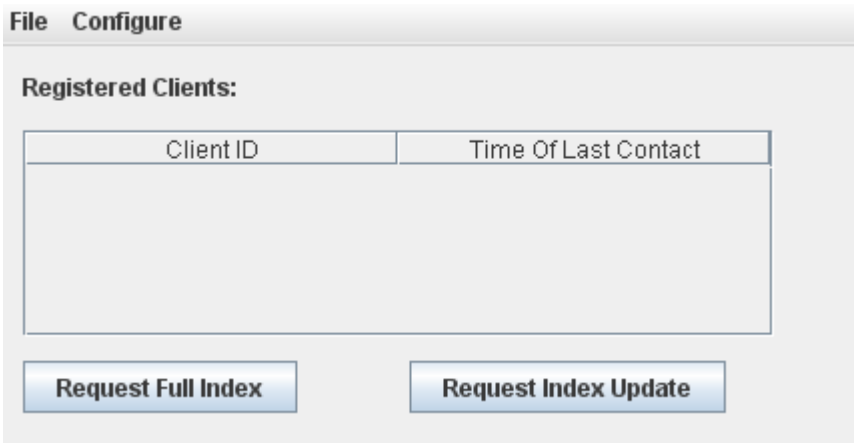

Figura 41: Interface de um *super peer*